# Understanding MicroSERVICE Architecture with Java & Spring Boot

**Rajeev Gupta**
**Java Trainer & Consultant**

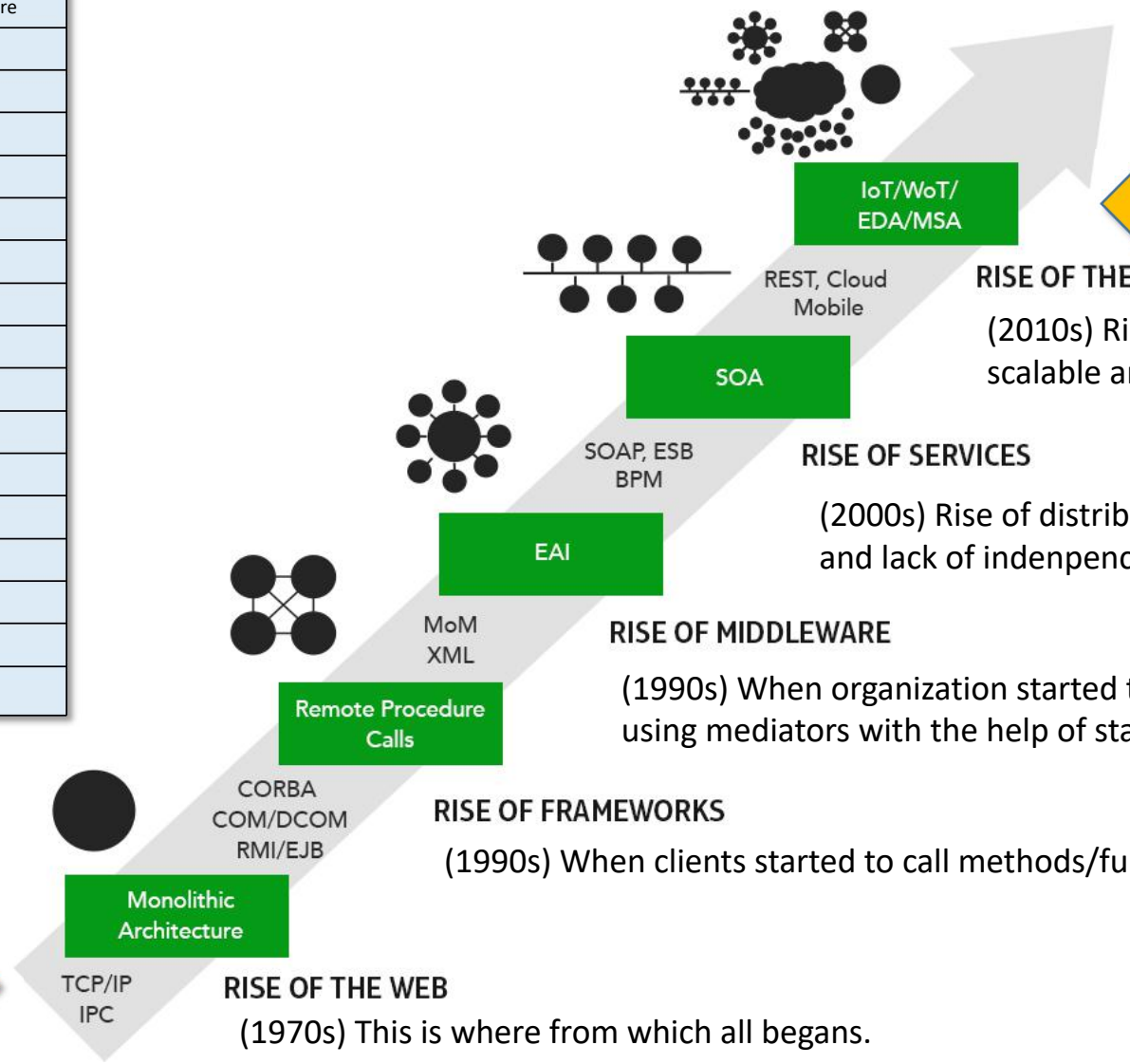
# Lets START ...

## Introduction

Lets start our journey through the deep space of microservice architecture, and explore what it provides, what it requires, and what are the underlying challenges.

# The architecture journey over the years ...

| | |
|---|---|
| CORBA | Common Object Request Broker Architecture |
| EJBs | Enterprise JavaBeans |
| RMI | Remote Method Invocation |
| COM | Component Object Model |
| DCOM | Distributed Component Object Model |
| EAI | Enterprise Application Integration |
| MoM | Management Object Model |
| XML | eXtensible Markup Language |
| SOA | Service Oriented Architecture |
| SOAP | Simple Object Access Protocol |
| ESB | Enterprise Service Bus |
| BPM | Business Process Model |
| IoT | Internet of Things |
| WoT | Web of Things |
| EDA | Event Driven Architecture |
| MSA | Microservices Architecture |
| REST | Representational State Transfer |




... and, in this presentation, we will gonna discuss this. **MSA (Microservice Architecture)**

**RISE OF THE MACHINES**

(2010s) Rise of consumer services that demand highly scalable and resilient distirbuted system.

**RISE OF SERVICES**

(2000s) Rise of distributed system with a lot of coupling, and lack of indenpence.

**RISE OF MIDDLEWARE**

(1990s) When organization started to integrate monolith enterprise applications using mediators with the help of standardized messaging formats and processes.

**RISE OF FRAMEWORKS**

(1990s) When clients started to call methods/functions on server.

**RISE OF THE WEB**

(1970s) This is where from which all begans.

It all started from here ...

https://www.appcentrica.com/wp-content/uploads/2016/10/microservices-insight-diagram.png

# What is a microservice architecture, anyway?

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


James Lewis


Martin Fowler

## Microservice architecture

- is-a Distributed application architecture >
  - composes-of small light weight self-contained services >
    - each-is stateless, self-discoverable,
    - implements a certain business capability or subdomain,
    - is independently maintainable, testable and deployable, and
    - provides theoretically endless scalability and resilience.

- is very different from a **monolithic architecture (the traditional design)**.
  - A monolithic architecture is
    - a single software entity that implements all the business concerns, capabilities or subdomains
    - highly cohesive and tightly coupled within and outside the system

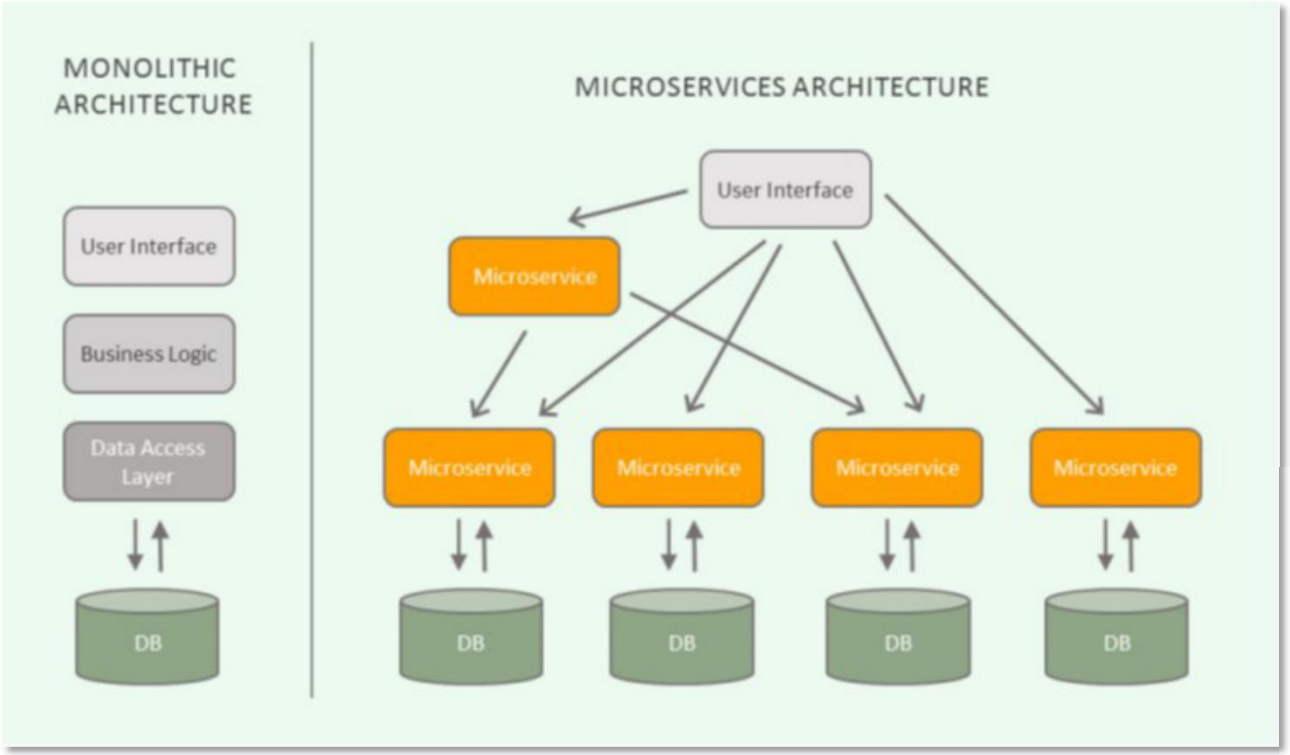
https://microservices.io/

Main portal that aggregates all the news and literature in one place.

# Microservice verses monolithic (traditional) architecture …

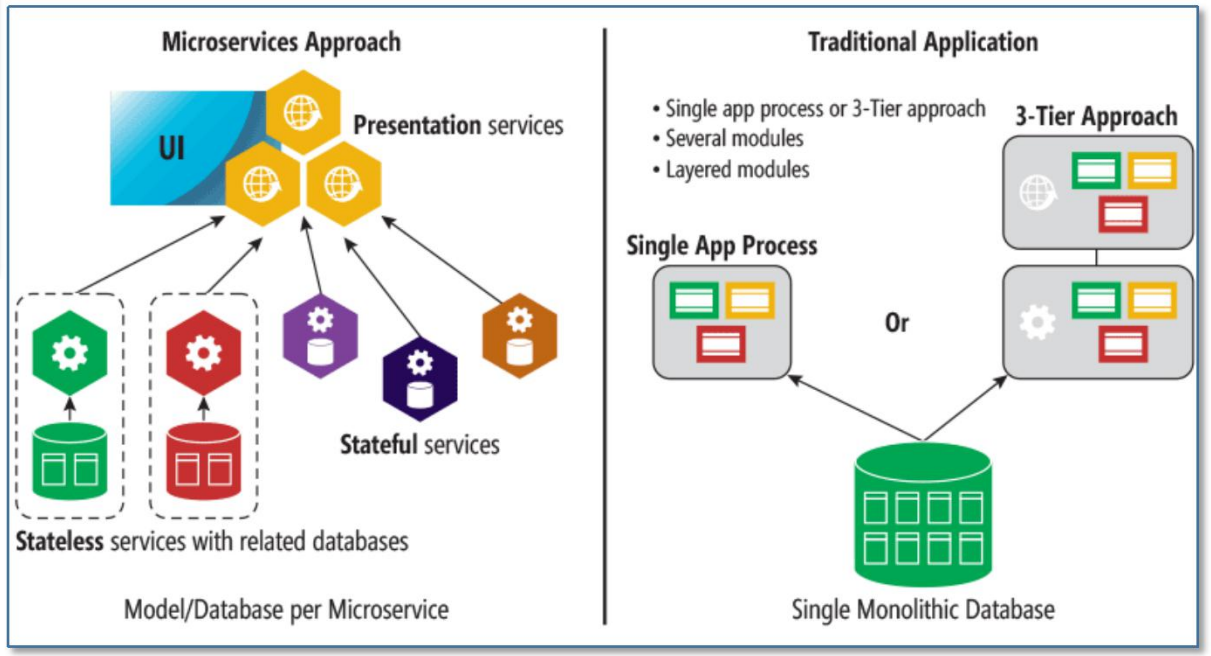
| Monolithic Architecture | Microservice Architecture |
|---|---|
| A single code base for all business goals. | Every unit of the entire application should be small, and it should be able to deliver one specific business goal. |
| Service startup takes more time. | Service startup is relatively quick. |
| Fault isolation is difficult. If any specific feature is not working, the complete system goes down. In order to handle this issue, the application needs to re-built, re-tested and also re-deployed. | Fault isolation is easy. Even if one service goes down, others can continue to function. |
| Monolithic architecture is tightly coupled. Changes in one module of code affects the other(s). | All microservices should be loosely coupled so that changes made in one does not affect the other. |
| Since services are not isolated, individual resource allocation not possible. | Businesses can deploy more resources to services that are generating higher ROI. |
| Application scaling is challenging as well as wasteful. | More hardware resources could be allocated to the service that is frequently used. For example in the e-commerce example, the number of users checking the product listing and doing search is large as compared to payments. So, more resources could be allocated to the search and product listing microservice. |
| Development tools get overburdened as the process needs to start from the scratch. | Microservices always remains consistent and continuously available. |
| Data is centralized. | Data is federated. This allows individual microservice to adopt a data model best suited for its needs. |
| Large team and considerable team management effort is required. | Small Focused Teams. Parallel and faster development. |
| Change in data model affects the entire database. | Change in the data model of one microservice does not affect other microservices. |
| Interaction is local and usually through method calls. | Interacts with other microservices by using well-defined interfaces. |
| Put emphasize on the entire project. | Microservices work on the principle that focuses on products, not projects. |
| One function or program depends on others. | No cross-dependencies between code bases. You can use different technologies for different microservices. |

# (cont.) Microservice verses monolithic (traditional) architecture …




https://www.edvantis.com/wp-content/uploads/2017/03/micros1.jpg

https://i.pinimg.com/originals/03/19/8f/03198fb151de525f30d5d1073dfa039e.png



# Microservice architecture promotes “easy manageability”. How?

Easy manageability by ...

- Having modules that can be
  - → designed and implemented in any order
  - → developed and maintained by separate small teams
  - → tested independently
  - → deployed independently
  - → easily evolve over time with more features, enhancements, and bug fixes
  - → made scalable and resilient with the help of infrastructure setup only

- Having plug and play design which means
  - → modules can be upgraded, replicated without making zero change impact on each others

- Having automation that provides
  - > **continuous integration** → providing new and better version of the module when available
  - > **continuous testing** → executing code static analysis, dynamic analysis, functional and integration testing, and failing builds that do not made up to the required standard
  - > **continuous deployment** → providing deployment of new builds which will also include upgrading the existing ones
  - > **continuous monitoring** → aggregating module execution logs, it performance and health metrics and doing analysis over them to determine system wide issues

# Microservice architecture, a kind of distributed architecture …

**Common design patterns for distributed systems:**

- → Communication orchestration
- → Externalizing configuration
- → Service discovery
- → Load balancing
- → Service orchestration
- → Circuit breakers
- → Intelligent routing
- → Micro-proxy
- → Control bus
- → One-time tokens
- → Global locks
- → Leadership election
- → Distributed sessions
- → Cluster state
- → Centralized accounting

- Microservice architecture ... a special type of distributed system design that
  - > enforces the use of common design patterns in distributed systems
  - > enables coordination leading to bioler plate patterns,
  - > allows developers to quickly stand up services and applications
    - that are capable to work well in any distributed environment

- With microservice architecture ...
  - > developer gets free from all the concerns of distributed system, and the coordination in between
  - > developer focuses more on the business concerns
  - > all the scalability and resilience requirements are handled externally and/or separately
  - > all the communication between the services becomes standardized as they started to follow known design patterns
  - > developers can work on separate services with no dependencies, and hence improving their productivity
  - > the design follows the best practices as they are inherently part of the implemented design patterns

# What are the characteristics for a microservice architecture?

- The architecture provides a **stable design** to the distributed system.
- It promotes **design for failures**.
  - Means it raises functional requirements for each possible type of failures in a distributed system.
- It enables **endless theoretical scalability and resilience**.
- There is no centralized governance means it has **decentralized governance**.
- It allows **endless evolution** to enable new or changes in business concerns, increament in scalability, availability and resiliance.
- The architecture promotes **smart endpoints with dumb pipes**.
- The architecture is usually based on **infrastructure automation** that includes CI, CD, CT and CM.
- It compose of services that are usually organized around **business concerns**.
- Each service is treated as **products rather than projects**.
- Services are design to be **self-cohesive**.
  - A service should implement a small set of strongly related functions.
  - Means like OOD, it should follow SRP (**Single Responsibility Principle**).
- Each service conforms to the CCP (**Common Closure Principle**).
  - Means things that change together should be packaged together - to ensure that each change affect only one service.
- All the **services are loosely coupled**.
  - Means each service as an API that encapsulates its implementation. The implementation can be changed without affecting clients.
- All services are designed to be **independently testable**.
  - Means although it is part of a business flow, but it should tested if it is being provided with all the required data.
- Each service is small enough to be **developed by a small team**, i.e. a team of 6-10 people.
- Each service is being owned by a separate completely **autonomous team**.
  - A team must be able to develop and deploy their services with minimal collaboration with other teams.

# What about EJBs? Are they better than microservice architecture?

# NO.

**Because ...**

- EJB requires sophisticated container that can manage enterprise beans life cycles
- EJB containers are bulky and have big footprints and hence cannot be use in a way to implement dynamic scalability
- EJB enforces Java tech stack on all the interacting services
- EJB interfaces are very cohesive and create highly coupled integrations
- EJB application are not single focused, and can be thought as a monolithic or microlithic application
- EJB development and testing cannot be made independent
- ... and others ...

Whereas ... microservice architecture has no such issues as mentioned above, and thus is more superior to EJB design of distributed system.

**What's EJB ?**

Enterprise JavaBeans (EJB) is one of several Java APIs for modular construction of enterprise software. EJB is a server-side software component that encapsulates business logic of an application.

EJB provides a mechanism to design and develop distributed system that provides a neat isolation of the server side logic that can be easily interface with the client side. Moreover it provides all the support and implementation that a distributed architecture requires.

An EJB web container provides a runtime environment for web related software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services. The EJB specification is a subset of the Java EE specification.

# What about SOAs? Are they better than microservice?

Again **NO** ... Following are some major differences between the two types of distributed systems:

| Concerns | Service over architecture (SOA) | Microservice architecture |
|---|---|---|
| **Composition** | It can be a collection of monolith or smart apps connected with ESB or intelligent message queues. | It is a collection of stateless self contained services connected through common interfaces or dump message queues. |
| **Design type** | Software components are integrated with smart pipes, or service buses. | Microservics are integrated with light weight interfaces like REST APIs, dumb message queues, etc. |
| **Nature of the application** | Monolithic in nature. | Full stack in nature. |
| **Coupling level** | The components are designed with higher cohesion. | The components are design with least or no cohesion. |
| **Independence** | The components cannot be operated or deployed independently. | The components can be operated or deployed independently. |
| **Dependency** | Business units are dependent. | They are independent of each other. |
| **Size of the Software** | Software size is larger than any conventional software. | The size of the Software is always small in microservices. |
| **Technology Stack** | The technology stack is lower compared to microservice. | Microservice technology stack could be very large |
| **Independent and focus** | SOA applications are built to perform multiple business tasks. | They are built to perform a single business task. |
| **Deployment** | The deployment process is time- consuming. | Deployment is straightforward and less time-consuming. |
| **Cost - effectiveness** | Less cost-effective. | More cost-effective. |
| **Scalability & resilience** | Scalability and resilience is not in the design; hence implemented explicitly. | Enforces scalability and high resilience by design. |
| **Single point of failure** | ESB could becomes the SPF. | No SPF in the design pattern, but can be implemented. |
| **Data sharing** | The components usually share data. | The components only share data when required. |
| **Testing** | Independent testing is mostly difficult. | Independent testing is very much possible. |

# What are the apparent challenges in microservice architecture?

- MicroServices rely on each other, and they will have to communicate with each other.
- Compared to monolithic systems, there are more services to monitor which are developed using different programming languages.
- As it is a distributed system, it is an inherently complex design, means there will be more ways to failures.
- Different services will have its separate mechanism, resulting in a large amount of memory for an unstructured data.
- Effective management and teamwork required to prevent cascading issues.
- Reproducing a problem will be a difficult task when it's gone in one version, and comes back in the latest version.
- Independent Deployment is complicated with microservices.
- Microservice architecture brings plenty of operations overhead.
- It is difficult to manage application when new services are added to the system.
- A wide array of skilled professionals is required to support heterogeneously distributed microservices.
- Microservice is costly, as you need to maintain different server space for different business tasks.

## COST of microservice architecture ...

1. **Operational Complexity**: You need a mature operations team to manage lots of services, which are being redeployed regularly

2. **Eventual Consistency**: Maintaining strong consistency is extremely difficult for a distributed system, which means everyone has to manage eventual consistency.

3. **Distribution**: Distributed systems are harder to program, since remote calls are slow and are always at risk of failure.

# Who are using microservice architecture?

A lot of companies and organizations have started using this architecture design. There are some companies which are using it before the word “microservice architecture” started to make headlines.

Following is the list of some companies ...

- Comcast Cable
- Uber
- Netflix
- Amazon
- Twitter
- IBM's Bluemix
- GE's Predix
- TheGuardian
- PayPal
- Gilt
- Zalando
- Ebay
- Sound Cloud
- Karma
- Groupon
- Hailo
- Capital One
- Lending Club
- AutoScout24
- Nordstrom
- ...

Each one of these companies have shared their success stories regarding how they have saved themselves by designing their software systems with microservice architecture. Following are some success stories ...

- https://qconnewyork.com/ny2015/presentation/partial-failures-microservices-jungle-survival-tips-comcast
- https://eng.uber.com/soa/
- http://techblog.netflix.com/
- http://highscalability.com/amazon-architecture
- http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf
- https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith
- https://blog.yourkarma.com/building-microservices-at-karma
- https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/
- https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-1/
- https://qconnewyork.com/ny2015/presentation/microservices-and-art-taming-dependency-hell-monster
- http://www.infoq.com/news/2016/02/Monolith-Microservices-Zalando
- http://www.capitalone.io/blog/why-capital-one-is-at-aws-re-invent-2015/
- http://neo4j.com/blog/managing-microservices-neo4j/
- http://www.infoq.com/news/2016/02/autoscout-microservices

# Designing by microservice architecture ...

**Designing ...**

Designing by microservice architecture takes a lot of efforts and bring a ton of changes in the engineering process with lots of activities and tasks. But in the end, it promises to deliver a distributed system that can scale with high resilience to the levels not possible otherwise.

# Digitizing business ...

**Microservice architecture provides a mechanism for digitizing a business ... by identifying services or application components that maps to business domains/sub-domains ... with communication flows mimicing the physical one.**

The services or components must be small enough for scalable deployment, and yet has narrow focus on certain business concern.

The modularization of the business into services must be done in a way that fulfills distributed computing requirements and facilitate high productivity and easy manageability ... which is only possible if the collection of services follow the business blueprints.

Desicision has to be made for each service regarding what to lose; Consistency, Availability, or Partition tolerance -- The CAP Theorem, by Eric Brewer.
Usually consistency is compromised for high scalabilty and resilience.

# How a business is decompose into microservice architecture?

- There are two strategies to decompose a system:

  1. Define services corresponding to **business capabilities**.
     - Means creating services for each business function; where a function captures “what” a business does or can do.
     - An organization’s business capabilities are identified by analyzing the organization’s purpose, structure, business processes, and areas of expertise.
     - It follows the Conway's Law ... which states:

       “Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.” -- Melvin Conway 1967

       → The law is based on the reasoning that in order for a software module to function, multiple authors must communicate frequently with each other. Therefore, the software interface structure of a system will reflect the social boundaries of the organization(s) that produced it, across which communication is more difficult.

  2. Define services corresponding to **business domain**.
     - Means creating services for each subdomain from the knowledge domain to which the business belongs.

- Both enables:
  - → Stable architecture since the business capabilities or subdomains are relatively stable.
  - → Development teams are cross-functional, autonomous, and organized around delivering business value rather than technical features.
  - → Services are cohesive and loosely coupled.

# Statelessness ...

**Statelessness → A core characteristics for any microservice.**

Means ... The service will be design with no assumptions of performing application/process/logic/data state management or handling as part of service function.

Hence ... The service is being designed such that,

- if it gets crashed, terminated abruptly, there won't be any loss of state or data except the processing of the current requests
- its multiple instances can be launch in parallel, and they will not need to be aware of each other (siblings)
- there will be no synchronization requirements between its instances
- any instance can be brought down any time without any need to backup any state or service data

**Hence ... it is very important to have stateless services.**
If states are required in a process, determine the state employing even sourcing pattern.

**With stateful design ...**

- makes a service very sensible to failures → as any failure will also lead to state/data loss
- creates service liability → as others will be depending on this service for accessing the state information, increasing coupling
- makes a service unexpandable → means it can not simply tear down as it is now hold state information and or data
- makes a service to have state synchronization process → so that it can synchronizes its states with its sibling instances
- makes a service to make sure of state consistency → so that each and every sibling instance has the same state at a given time
- makes a service inefficient → as it has to manage the state/data, sync with other instances, and make sure of its consistency
- makes a service incompatible with its other (older or newer) versions → as any change in service may have impact on the state representation, sync process, or consistency strategy/assumptions
- makes a service hard to upgrade → as will require all the instances to upgrade as once

# How a service in the microservice architecture should be scoped?

Scoping a microservice means, determining the functional requirements, its computation, communication, and I/O needs.

### Pick a purpose

Pick a business concern, capability or a sub-domain.

### Adequate patterns

Use adequate communication and data patterns to support max scalability.

### Ensure scalability

Make sure it can collaborate with its copies.

### Identify data

Identify local, private, shareable, and public data …

- Local: Data that is specific to the service instance.
- Private: Data that is specific to the service, but shareable by all instances.
- Shareable: Data that can be shared with other services.
- Public: Data that can be exposed to the external entities.

### Eventual consistency

If there are many services working on a state, then make sure the overall system remains eventually consistent → Use event sourcing pattern.

### Ensure statelessness

Make sure its operations remains state-less

- Means the service MUST NOT save data as local which is private, shareable or public.

### Ubiquitous comm.

Use ubiquitous communicate dialogues.

### High frequency I/O

If there are enormous reads and writes that affects each other value streams' efficiencies, then employ CQRS pattern.

### Resilience patterns

Use all the required resiliency patterns to implement fault tolerance within the service.

### Logging & monitoring

Implement proper logging system with service monitoring that also includes either exposing or sending performance metrices.

# How data should be persisted by a microservice?

A microservice must deal with the data and state with certain guidelines. These guidelines enables easy data manageability in a microservice architecture.

**The guidelines ...**

1. The service must never persists any data within its own environment/context/container.
   → As the locally saved data can be lost if the service environment/container is reinitiated or recreated.
2. The service must never persists any state, it should be determine from the events history/log on demand.
3. If two or more services need to work on a state, then use event sourcing design pattern (discuss later).
4. The service must use the best storage strategy (discuss later) based on the type of data and its usage requirements.
5. The service should consider CQRS design pattern (discuss later) for entities impacting system performance due to their high volume access.
6. If a service creates an entity record/object/instance, it must also own the entity lifecycle, and provides APIs to other services that need to access this entity records.
   → Means, an entity record must not be updated directly by multiple services. One one must own the entity that creates its records. The other services access the entity through the owning service's APIs.
7. The persistent store should also be configured and deployed as cluster to prevent SPF due its errors.
8. The persistent store access information and credentials must be store in central configuration for all service instances.

# The data persistent requirements in a microservice …

| Type of data | Usage Requirements | Storage Strategy | Tools | Example |
|---|---|---|---|---|
| Hierarchical data | CRUD operations | SQL based storage | MySQL, PostgreSQL | User database with their professions, and association data |
| Large sized hierarchical data | Mostly reads & append operations | Doc based NoSQL storage | MongoDB | User feeds with associated content, and engagements |
| Large sized non-hierarchical data | Mostly reads & append operations | Key-value based NoSQL storage | Cassandra, DynamoDB, Redis | User activities |
| Temporary data | High speed read and write access | Cache based storage | MemCached, Redis | User session, notifications |
| Command, requests, events | Produced, consumed separately asynchronously | Message queue | Kafka, ActiveMQ | User messages to other user, OR user yet to receive notifications |
| Data records, objects | Produced periodically and/or on each operation | Aggregating operational logs/records | Logstash, Fluentd | User engagements, activities |

# Microservice architecture example | Hotel Management System

Imagine a 5 star hotel, that requires a system to manage its operations.

**Business Entities ...**

| | |
|---|---|
| Guest | Staff |
| Booking | Room |
| Service | Bill |
| Payment | |

**Business Concerns ...**

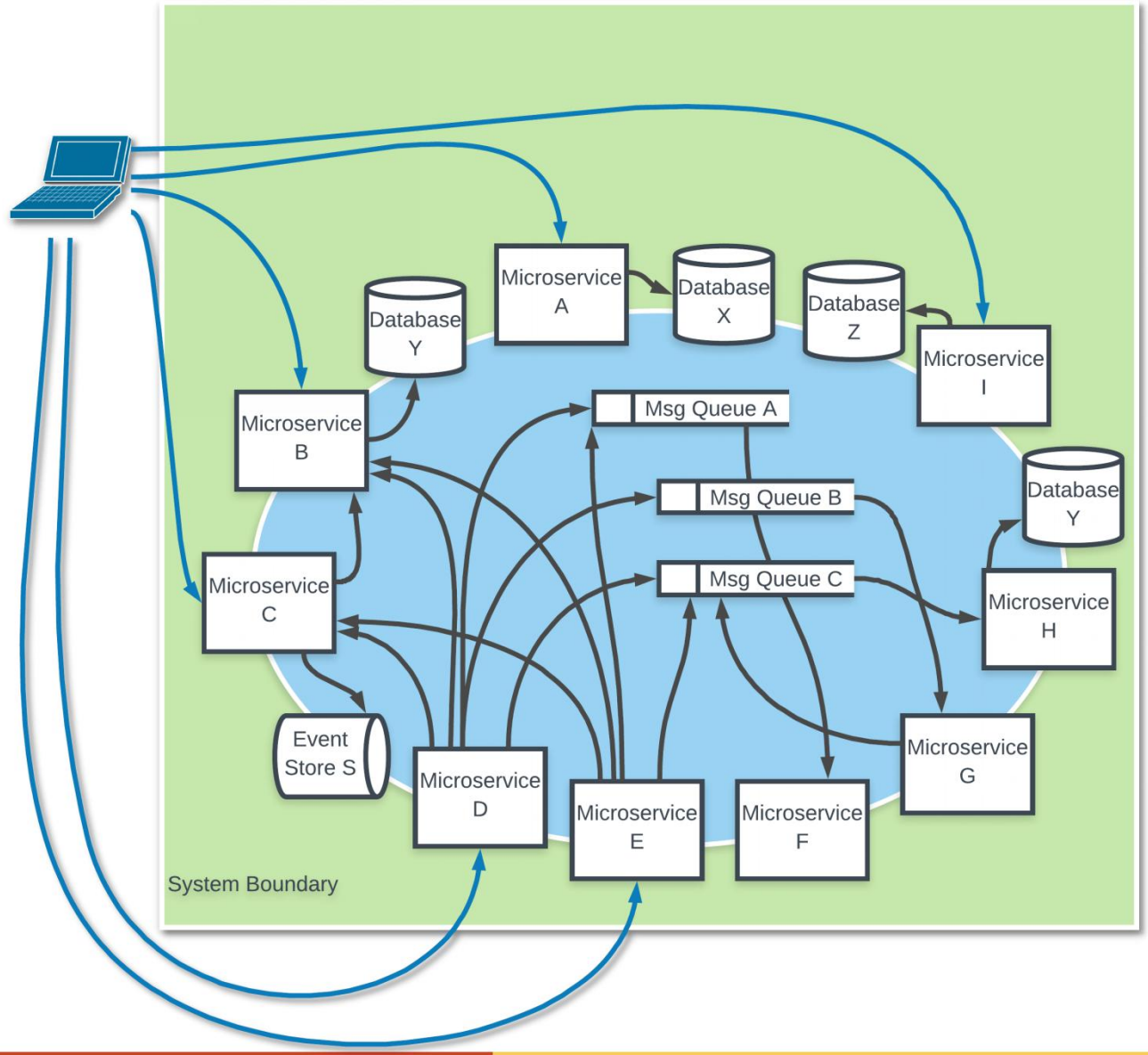
- Customer onboarding
- Booking a room
- Room maintenance
- Checking in
- Room service
- Checking out
- Billing customer
- Payment order

**Microservices ...**

| | |
|---|---|
| • **Customer signup / onboarding service** | Responsibility: Manages customers; contains onboarding process<br>Owns: Customer database |
| • **Room searching service** | Responsibility: Manages hotel room schedule<br>Owns: hotel's room availability schedule database |
| • **Booking creation service** | Responsibility: Manages customers booking; manages booking states<br>Owns: Booking event store |
| • **Checkin service** | Responsibility: Executes checkin process<br>Triggers: Room searching, maintenance, room-service, billing services |
| • **Checkout service** | Responsibility: Executes checkout process<br>Triggers: Room searching, room maintenance, billing services |
| • **Room maintenance service** | Responsibility: Execute periodic and ondemand maintenance cycle<br>Triggers: Room searching service |
| • **Room service** | Responsibility: Execute periodic or ondemand service process<br>Triggers: Billing service |
| • **Billing service** | Responsibility: Manages customers billing<br>Owns: Customer billing database |
| • **Payment service** | Responsibility: Manages customers payments<br>Owns: Customer payment database |

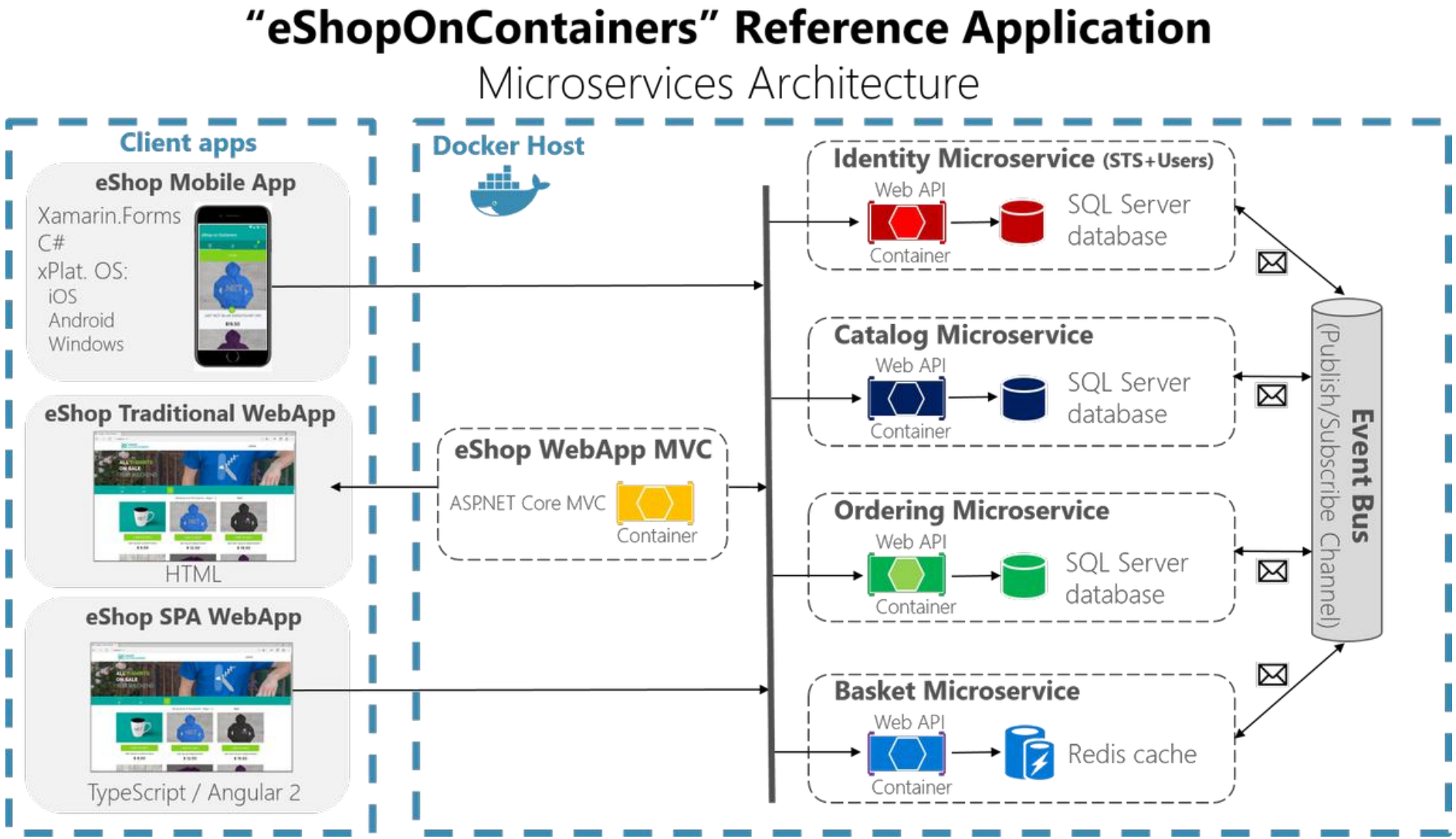
# Microservice architecture example | (cont.) Hotel Management System

The architectural diagram will be ...




| | |
|---|---|
| Microservice A | Customer signup / onboarding service |
| Microservice B | Room searching service |
| Microservice C | Booking creation service |
| Microservice D | Checkin service |
| Microservice E | Checkout service |
| Microservice F | Room maintenance service |
| Microservice G | Room service |
| Microservice H | Billing service |
| Microservice I | Payment service |
| Database X | Customers database |
| Database Y | Billing database |
| Database Z | Payment database |
| Event Store S | Booking events store |
| Msg queue A | Room maintenance msgs |
| Msg queue B | Room service msgs |
| Msg queue C | Billing msgs |

# Microservice architecture example | eCommerce Store




https://pbs.twimg.com/media/DAKIFlmUAAAh2sS.png

# What are the major tech stack for microservices?

A microservice can be developed in any of the tech stack that provides the necessary requirments (as mentioned before).

Popular Tech Stacks ...

Other Tech Stacks ...


**Oracle's Java ...**
Spring Boot, Spring Cloud, Netflix OSS, Play, GWT, Spark, Micronaut, Javalin, JHipster, ...

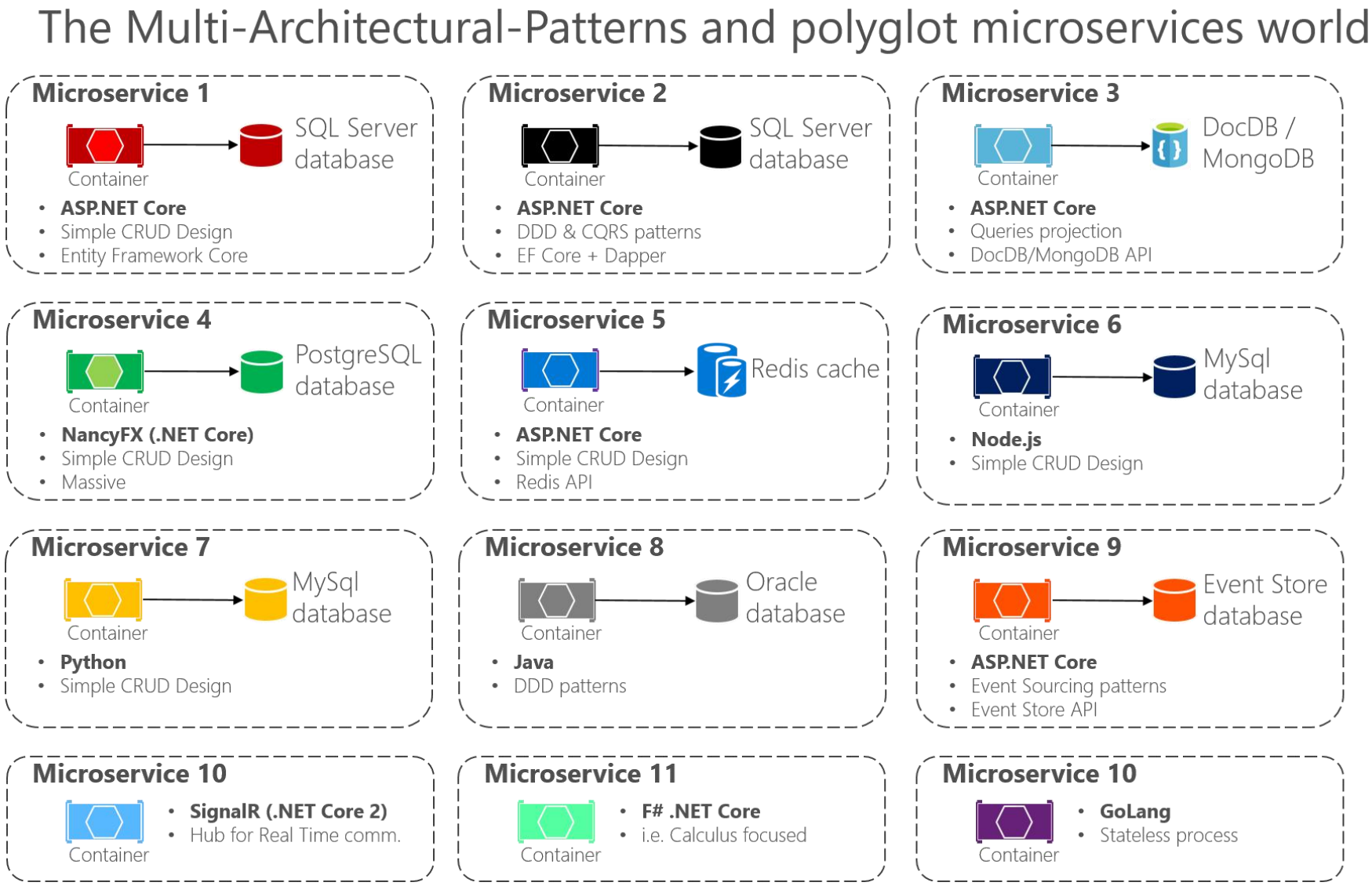

**JavaScript ...**
Node.js, Express, Molecular, Seneca, Sails, Loopback, Koa, Devis, ...




**Microsoft's .Net framewok ...**
Service Fabric, .Net Core, OWIN, Steeltoe, ServiceStack, ...

**Python ...**
Flask, Nameko, appkernel, ...

# Heterogenous tech stack example in a microservice architecture ...




https://png2.kisspng.com/20180426/vfe/kisspng-architectural-pattern-architecture-microservices-s-simple-patterns-5ae22420b11bb8.933574741524769824 7254.png


For the rest of the presentation, we will be only concerned and focused on **Java** based technology frameworks especially Spring framework, Spring Boot, Spring Cloud Services and Netflix Open Source Softwares.


NETFLIX
OSS

# What is Spring framework?

https://spring.io/projects/spring-framework

The Spring Framework ...

- is an application framework and inversion of control container for the Java platform
  - with features that can be used by any Java application,
  - but there are extensions for building web applications on top of the Java EE platform.
- does not impose any specific programming model, and has become popular in the Java community as an addition to, or even replacement for the Enterprise JavaBeans (EJB) model.
- is open source.

**Provides ...**

- Dependency Injection
- Proxy Pattern
  - → Dynamic proxy
  - → CGLib Proxy
- Transaction Management
- Aspect Oriented Programming
- Lazy class loading

**The Spring Framework includes several modules that provide a range of services:**

- **Spring Core Container**: this is the base module of Spring and provides spring containers (BeanFactory and ApplicationContext).
- **Aspect-oriented programming**: enables implementing cross-cutting concerns.
- **Authentication and authorization**: configurable security processes that support a range of standards, protocols, tools and practices via the Spring Security sub-project (formerly Acegi Security System for Spring).
- **Convention over configuration**: a rapid application development solution for Spring-based enterprise applications is offered in the Spring Roo module
- **Data access**: working with relational database management systems on the Java platform using Java Database Connectivity (JDBC) and object-relational mapping tools and with NoSQL databases
- **Inversion of control container**: configuration of application components and lifecycle management of Java objects, done mainly via dependency injection
- **Messaging**: configurative registration of message listener objects for transparent message-consumption from message queues via Java Message Service (JMS), improvement of message sending over standard JMS APIs
- **Model–view–controller**: an HTTP- and servlet-based framework providing hooks for extension and customization for web applications and RESTful (representational state transfer) Web services.
- **Remote access framework**: configurative remote procedure call (RPC)-style marshalling of Java objects over networks supporting Java remote method invocation (RMI), CORBA (Common Object Request Broker Architecture) and HTTP-based protocols including Web services (SOAP (Simple Object Access Protocol))
- **Transaction management**: unifies several transaction management APIs and coordinates transactions for Java objects
- **Remote management**: configurative exposure and management of Java objects for local or remote configuration via Java Management Extensions (JMX)
- **Testing**: support classes for writing unit tests and integration tests

-- from Wikipedia

# What is Spring Boot?

https://spring.io/projects/spring-boot

A microservice is a

- self-contained, easily deployable, business focused, and an independent application ... that
  - owns its produced data, operates statelessly, easily discoverable, and works with remote configuration.

In Java, one can create a self-contained application using **Spring Boot framework**.

- It is based on Spring framework with ease of application configuration using annotations
  - Means ... No bulky YAML or XML spring configuration files
- Embed **Tomcat**, **Jetty** or **Undertow** directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

- **Spring Cloud Services** and **Netflix OSS** are the collection of frameworks and tools that allows a spring boot application to become a part of a microservice architecture

Spring Cloud

Spring Boot

# What is Spring Cloud Services?

https://spring.io/projects/spring-cloud

- Spring Cloud provides tools for developers to quickly build some of the common patterns (mentioned previously) in distributed systems.
- Coordination in a distributed system leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns.
- They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms like GCP, AWS, CloudFoundary, etc.
- Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.
- Spring Cloud takes a very declarative approach, and developers often get a lot of fetaures with just a classpath change and/or an annotation.
- Spring cloud services has a lot of projects ... some of the important ones are as follows:

| | | |
|---|---|---|
| • Spring Cloud Config | • Spring Cloud Connectors | • Spring Cloud Cluster |
| • Spring Cloud Netflix | • Spring Cloud Sleuth | • Spring Cloud Consul |
| • Spring Cloud Security | • Spring Cloud Stream | • Spring Cloud Zookeeper |
| • Spring Cloud Gateway | • Spring Cloud Stream App Starters | • Spring Cloud Cloudfoundry |
| • Spring Cloud OpenFeign | • Spring Cloud Task | • Spring Cloud AWS |
| • Spring Cloud Data Flow | • Spring Cloud Task App Starters | • Spring Cloud Starters |
| • Spring Cloud Function | • Spring Cloud Contract | • Spring Cloud CLI |
| • Spring Cloud Bus | • Spring Cloud Open Service Broker | • Spring Cloud Pipelines |

**Features ...**

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- API gateway
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging
- Event, and message bus
- Support for Kubernetes, AWS, and CloudFoundary
- Etc...

# What are Spring Boot projects/modules?

| | |
|---|---|
| Spring Cloud Config | Centralized external configuration management backed by a git repository. The configuration resources map directly to Spring Environment but could be used by non-Spring applications if desired. |
| Spring Cloud Netflix | Integration with various Netflix OSS components (Eureka, Hystrix, Zuul, Archaius, etc.). |
| Spring Cloud Security | Provides support for load-balanced OAuth2 rest client and authentication header relays in a Zuul proxy. |
| Spring Cloud Gateway | Spring Cloud Gateway is an intelligent and programmable router based on Project Reactor. |
| Spring Cloud OpenFeign | Spring Cloud OpenFeign provides integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. |
| Spring Cloud Data Flow | A cloud-native orchestration service for composable microservice applications on modern runtimes. Easy-to-use DSL, drag-and-drop GUI, and REST-APIs together simplifies the overall orchestration of microservice based data pipelines. |
| Spring Cloud Function | Spring Cloud Function promotes the implementation of business logic via functions. It supports a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS). |
| Spring Cloud Bus | An event bus for linking services and service instances together with distributed messaging. Useful for propagating state changes across a cluster (e.g. config change events). |
| Spring Cloud Connectors | Makes it easy for PaaS applications in a variety of platforms to connect to backend services like databases and message brokers (the project formerly known as "Spring Cloud"). |
| Spring Cloud Sleuth | Distributed tracing for Spring Cloud applications, compatible with Zipkin, HTrace and log-based (e.g. ELK) tracing. |
| Spring Cloud Stream | A lightweight event-driven microservices framework to quickly build applications that can connect to external systems. Simple declarative model to send and receive messages using Apache Kafka or RabbitMQ between Spring Boot apps. |
| Spring Cloud Stream App Starters | Spring Cloud Stream App Starters are Spring Boot based Spring Integration applications that provide integration with external systems. |
| Spring Cloud Task | A short-lived microservices framework to quickly build applications that perform finite amounts of data processing. Simple declarative for adding both functional and non-functional features to Spring Boot apps. |
| Spring Cloud Task App Starters | Spring Cloud Task App Starters are Spring Boot applications that may be any process including Spring Batch jobs that do not run forever, and they end/stop after a finite period of data processing. |
| Spring Cloud Contract | Spring Cloud Contract is an umbrella project holding solutions that help users in successfully implementing the Consumer Driven Contracts approach. |
| Spring Cloud Open Service Broker | Provides a starting point for building a service broker that implements the Open Service Broker API. |
| Spring Cloud Cluster | Leadership election and common stateful patterns with an abstraction and implementation for Zookeeper, Redis, Hazelcast, Consul. |
| Spring Cloud Consul | Service discovery and configuration management with Hashicorp Consul. |
| Spring Cloud Zookeeper | Service discovery and configuration management with Apache Zookeeper. |
| Spring Cloud Cloudfoundry | Integrates your application with Pivotal Cloud Foundry. Provides a service discovery implementation and also makes it easy to implement SSO and OAuth2 protected resources. |
| Spring Cloud AWS | Easy integration with hosted Amazon Web Services. It offers a convenient way to interact with AWS provided services using well-known Spring idioms and APIs, such as the messaging or caching API. Developers can build their application around the hosted services without having to care about infrastructure or maintenance. |
| Spring Cloud Starters | Spring Boot-style starter projects to ease dependency management for consumers of Spring Cloud. (Discontinued as a project and merged with the other projects after Angel.SR2.) |
| Spring Cloud CLI | Spring Boot CLI plugin for creating Spring Cloud component applications quickly in Groovy |
| Spring Cloud Pipelines | Spring Cloud Pipelines provides an opinionated deployment pipeline with steps to ensure that your application can be deployed in zero downtime fashion and easily rolled back of something goes wrong. |

# What is Netflix Open Source Softwares?

https://netflix.github.io/

## Some important & famous projects (among 150) ...

- **Zuul**: Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.
- **Eureka**: AWS Service registry for resilient mid-tier load balancing and failover.
- **Hystrix**: Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services
- **Ribbon**: Ribbon is a Inter Process Communication (remote procedure calls) library with built in software load balancers. The primary usage model involves REST calls with various serialization scheme support.
- **ChaosMonkey**: Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures.
- **SimianArmy**: Tools for keeping your cloud operating in top form. Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures.
- **Falcor**: A JavaScript library for efficient data fetching.
- **Genie**: Genie is a federated job execution engine which provides REST-ful APIs to run a variety of big data jobs like Hadoop, Pig, Hive, Presto, Sqoop, and more.
- **Inviso**: Inviso is an interface to search and visualize Hadoop jobs, Job performance, and cluster utilization data.
- **Aegisthus**: Aegisthus enables the bulk abstraction of data out of Cassandra for downstream analytic processing.
- **EVCache**: EVCache is a memcached and spymemcached based caching solution that is mainly used for AWS EC2 infrastructure for caching frequently used data.
- **Dynomite**: Inspired by Amazon's Dynamo whitepaper, Dynomite is a generic dynamo implementation for different k-v storage engines.
- **MSL**: Message Security Layer (MSL) is an extensible and flexible secure messaging framework that can be used to transport data between two or more communicating entities.
- **Astyanax**: Astyanax is a high-level Java client for Apache Cassandra.
- **Asgard**: Asgard is a web-based tool for managing cloud-based applications and infrastructure.

- **Netflix OSS is a set of frameworks and libraries**
  - **that Netflix wrote to solve some interesting distributed-systems problems at scale.**

- Today, for Java developers, it's pretty synonymous with developing microservices in a cloud environment.
- Patterns for service discovery, load balancing, fault-tolerance, etc are incredibly important concepts for scalable distributed systems and Netflix brings nice solutions for these.

- Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.
  - With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.

# Activities in system design with microservice architecture ...

## Primary activities ...

1. Identify business capabilities / subdomains / concerns.
2. For each business concern,
   a. Determine the interfaces and their types; API, RPC or msg queue.
   b. Determine what are the other interacting concerns with this one
   c. Determine the tech stack.
   d. Determine the data needs.
   e. Identify the team and size
3. Setup/create and deploy **discovery server**
4. Setup/create and deploy **configuration server**
5. Setup/create and deploy **API gateway** OR edge server
6. Setup and deploy databases required for various services
7. Setup and deploy message queues required for various services
8. For each identified concern
   a. Create a separate repo, and project
   b. Create the service itself such that
      i. It will fetch its configuration from the configuration server
      ii. It will register itself to the discovery server
      iii. Define routes for each APIs that will be public to client in the API gateway
      iv. Implement the service
   c. Use **swagger.io** or other similar tech to document APIs
   d. Determine whether this service will be using a state and there is a requirement for implementing **event sourcing**
   e. Determine the data access/update frequency, and if it is high then consider implementing **CQRS** for that data entity
   f. Setup CI tool for the service
   g. Setup CD tool for the service
   h. Determine the testing strategy and how it will be automated

## Secondary activities ...

1. Setup app **containerization** infrastructure
2. For each service
   a. Configure CI tool to generate containerized builds
   b. Configure CD tool to deploy containerized builds on containers
3. Setup **load balancers** for certain services that are expected of high load
4. Setup **reverse proxy** for the APIs if there is a need

## Optional activities ...

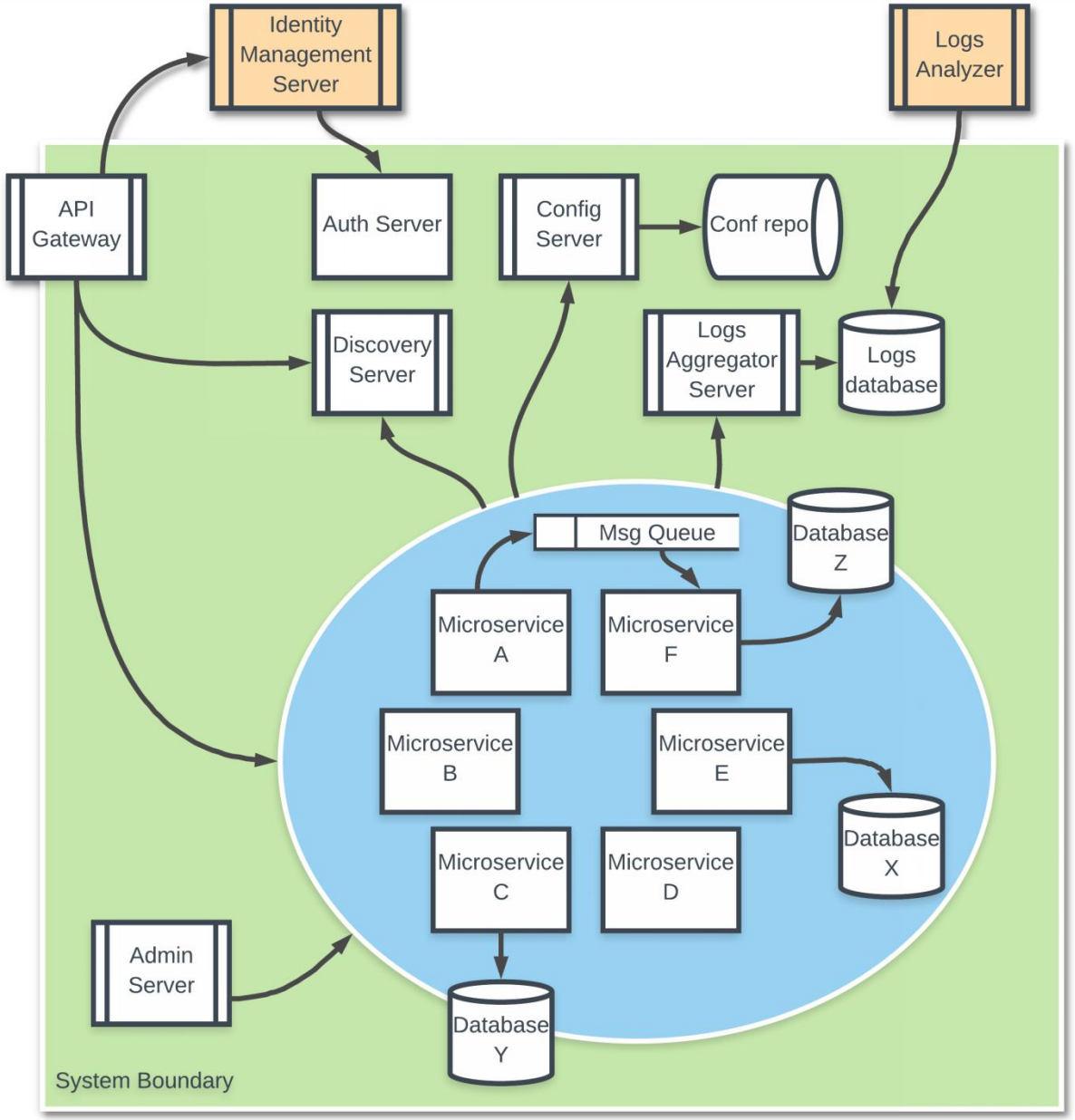
1. Setup **service tracing**
2. Setup **tracing log analyzer** service/server
3. Setup **logs aggregation** server
4. Setup **logs analyzer** servers
5. Setup **service monitoring** server
6. For each service
   a. Configure project to capture service call and send them to the tracing server
   b. Configure service logs to be send to a central logs server
   c. Enable enable **HATEAOS**
   d. Enable **monitoring actuators**
7. Define deployment policy for the containerization infrastructure

# (cont.) Activities in system design with microservice architecture ...

So,
From the activities, one might get an idea that ...

**Microservice architecture is not about the software design, but rather it engolfs all the engineering process from planning to final deployment and all the milestones inbetween.**

# Components of a microservice architecture …




**Components**

1. Business Microservices
2. API Gateway
3. Discovery Server
4. Configuration Server
5. Identity Management Server
6. Authentication Server
7. Logs Aggregator Server
8. Logs Analyzer
9. Admin Server
10. Databases
11. Message Queues
12. Load Balancers (not shown)
13. Reverse Proxies (not shown)

# (cont.) Components of a microservice architecture ...

## 1) Business Microservice ...

- This will be the self-contained application that has a single focus to a business concern.
- It can be implemented using any tech stack based on Java, .Net, JS, Python, or any other language.
  - For Java, one can easily use Spring Boot with packages from Spring Cloud and Spring Cloud Netflix.
- It should be containerized either using Dockers or any other tech.
- It must be designed and implemented in a way that allows deployment of its multiple instances.
- It must be stateless.

## 2) API Gateway ...

- This is a software component often uses third party frameworks to perform static and dynamic routing for all the business microservices.
  - In routing, it determine the target microservice, find out its physical address, and then relay the request.
- It may also implements session validations, and other calling filters requires for every call.
- It may also do request re-write before and after the routing process.

## 3) Discovery Server ...

- This is a service often available as third party software or framework. It provides service registry. All the microservices can register themselves, and anyone can determine the physical address by service name. It may also do load balancing for microservices getting registered with same name.

## 4) Configuration Server ...

- This is a microservice often available as third party software or framework. It provides configuration store to the business microservices. Hence services can get there config data from this server, and this relieve them from having this configuration as part of their package.

## 5) Identity Management Server ...

- This will be an external system that has knowledge about the users, and their authentication process. It provides the auth token that can be use to identify user's session.

# (cont.) Components of a microservice architecture ...

## 6) Authentication Server ...

- It is usually implemented as a microservice, and provides login, logout, and refresh APIs.
- It can implement any standard or custom protocol for authentication, and session creation.

## 7) Logs Aggregator Server ...

- It is usually a third party service that captures all the logs streaming from the business microservices.
- It transform these logs into structure records, and store them in a database.

## 8) Logs Analyzer ...

- It is a third party application that works on the logs aggregation server database, and based on the prepared views, generate reports and analytical data.

## 11) Message Queues ...

- These are third party systems. They implement messaging patterns and are used between multiple business microservices for message passing.

## 10) Databases ...

- These are third party system. Based on the data needs, they can be SQL, non-SQL, or simple fast caches. They are being deployed as a single entity, but in actual has replications in the form of cluster or pair, so as to provide fault tolerance, and load balancing.

## 9) Admin Server ...

- It is can be an external server or a custom build one. It takes performance metrics from all the business microservices, and have them display in a single dashboard. It can also generate compressed logs, reports and analytical data.

## 12) Load Balancers ...

- These are third party systems. works as a reverse proxy and spread incoming requests on all the instances of a business microservice.

## 13) Reverse Proxies / Side-Car Proxies ...

- These are third party systems that are deployed before the microservice to assist it in many different ways like filtering, protecting, or monitoring content.

# (cont.) Components of a microservice architecture ...

| Component ... | Technology ... | |
|---|---|---|
| Business microservice | Spring Boot with Spring framework modules, Spring Cloud Services, Spring Cloud Netflix ... | Implement a new microservice using Spring Boot project from https://start.spring.io/, and enable configuration server, eureka client, sleuth, and actuator |
| API Gateway | Netflix Zuul ... provided by Spring Cloud Netflix project. | Implement a new microservice using Spring Boot project and enable Zuul support. Also implement filters for JWT where OAuth2 token will get verified. |
| Discovery Server | Netflix Eureka server, Zookeeper, Consul, ... | Implement a new microservice using Spring Boot project and enable Eureka support. |
| Configuration Server | Spring Cloud Config | Implement a new microservice using Spring Boot project and enable Configuration support. Make sure to define bootstrap property for every business service. |
| Identity Management Server | KeyCloak | Install the server on a separate machine/instance, and configure it using its UI. The API gateway will be configure to contact KeyCloak on login/logout requests. |
| Authentication Server | Spring Security, Spring Boot Security, JWT, OAuth2 | Implement a new microservice using Spring Boot project and implement security configuration, context and other required beans. Also implement login, logout, and refresh APIs. |
| Logs Aggregator Server | Zipkin, Spring Cloud Sleuth, Logstash, Fluentd, DataDog, PinPoint, New Relic, etc | Make sure each and every business microservice has sleuth dependency added to the project. For the server, install the prebuild binary for the Zipkin server, and configure its access path in all the business services' application properties. |
| Logs Analyzer | Zipkin, Kibana, Splunk, PaperTrail, Timber.io, etc | Deploy prebuild Zipkin. Use its UI for the logs view. Save aggregated logs on ElasticSearch, and then use Kibana to view analysis. |
| Admin Server | Spring Boot Actuator, Spring Boot Admin Server, Spring Boot Admin Client | Create a new microservice with admin server enabled. Add required properties in the properties file. Make sure to add actuator and admin-client dependency in each business microservice, and define the admin server access path in each service properties. |
| Databases | SQL: MySQL, Oracle, PostgreSQL \| NoSQL: MongoDB, Cassandra, Redis, ... | Install and deploy binary package on the machine/instance. |
| Message Queues | Kafka, ActiveMQ, RabbitMQ, ... | Install and deploy binary package on the machine/instance. |
| Load Balancers | NginX, Seesaw, Neutrino, HAProxy, ... | Install and deploy binary package on the machine/instance. |
| Reverse Proxies / Side-Car Proxies | NginX, HAProxy, Squid, Vulcand, Apache Traffic Server, Isto, Linkerd ... | Install and deploy binary package on the machine/instance. |

# (cont.) Components of a microservice architecture ...

**So, in short →**

## 1) Develop infrastructure microservices ...

- This will involve creating new microservices for:
  1. API gateway → Zuul
  2. Discovery server → Eureka, Zookeeper or Consul
  3. Monitoring admin server → Spring Boot Admin
  4. Configuration server → Spring Boot Config or Consul
  5. Authorization server → Spring Cloud Security
- Creating new code repositories and projects for each service.
- Here admin server can be embded into discovery for easy monitoring.

## 5) Configure business microservices ...

- Create configuration GIT repository, and in an adequate directory structure, copy all the application.properties or yml files into the repo directory with proper names.
- Define common application.properties file for global properties.
- Setup configuration of the config service.
- Setup bootstrap.properties file for each and every microservice that includes infrastructure and business ones.
  - In this property file, define the discovery server settings, and the configuration name.

## 2) Setup remaining infrastructure services ...

- This will involve deploying and setting up services for:
  1. Identity manager server
  2. Reverse proxies
  3. Load balancers
  4. Logs aggregator services → Zipkin, Logstash or Prometheus
  5. Logs analyzers → Kibana or Grafana

## 3) Setup message queues and databases ...

- This will involve deploying and setting up message queues and databases.
- If possible, made them discoverable from the discovery service.
- If possible, made their configuration available from the configuration service.

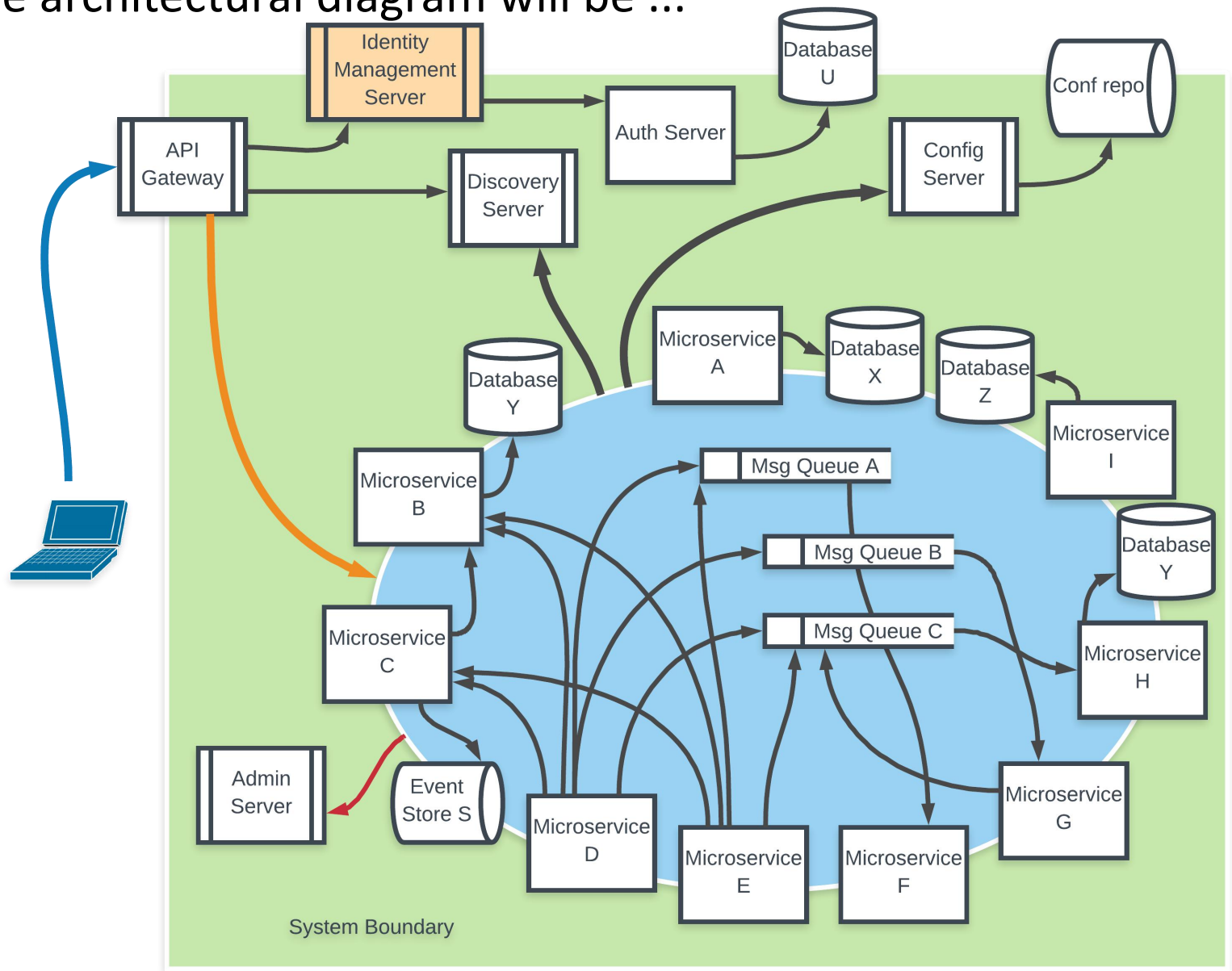
## 4) Develop business microservices ...

- Creating new code repositories and projects for each service.
- Developing each microservice as a standalone product.
- Test these microservices.

Once all of this is done, then work on the deployment plan.

# Microservice architecture example | Hotel Management System

The architectural diagram will be ...




Example continued from previous slide.

| | |
|---|---|
| Microservice A | Customer signup / onboarding service |
| Microservice B | Room searching service |
| Microservice C | Booking creation service |
| Microservice D | Checkin service |
| Microservice E | Checkout service |
| Microservice F | Room maintenance service |
| Microservice G | Room service |
| Microservice H | Billing service |
| Microservice I | Payment service |
| Database U | Users + customers database |
| Database X | Customers database |
| Database Y | Billing database |
| Database Z | Payment database |
| Event Store S | Booking events store |
| Msg queue A | Room maintenance msgs |
| Msg queue B | Room service msgs |
| Msg queue C | Billing msgs |

# Advance design patterns ...

**ES & CQRS**

**IF** the distributed system consists of services that requires same app/process/data state,
**OR** has certain business entities that reqiures high volume access,
**THEN** consider implementing event sourcing and CQRS design patterns.

# Event Sourcing | What is it?

A design pattern ... used to implement “eventual consistency” within a distributed system.

Without event sourcing ... a distributed system will undergo event processing issues like ...

- Ineffective or inconsistent event management
- Missing of events due to slow reaction time, slow consumer, faster producer
- Multiple modules working with application state endup with inconsistent states
- Waste of consumer resources due to events polling
- Inability to reconstruct current state after failover
- Synchronization requirements between engaging modules in order to enforce consistent state processing
- Etc.

**What is eventual consistency?**

- It is a type of **Relaxed Temporal Contraint** on the data/state consistency.
  - Means it enables a little compromise on the consistency until and unless the engaging parties conform to certain requirements.
- It guarantees the state of all participants to converge towards a consistent state at some point in the future.
- E.g. database cluster … All the replicas are maintained by coordinating writes between multiple nodes; making sure that writes must be coordinated to all replicas in the exact order that they were received.

## What is the solution?

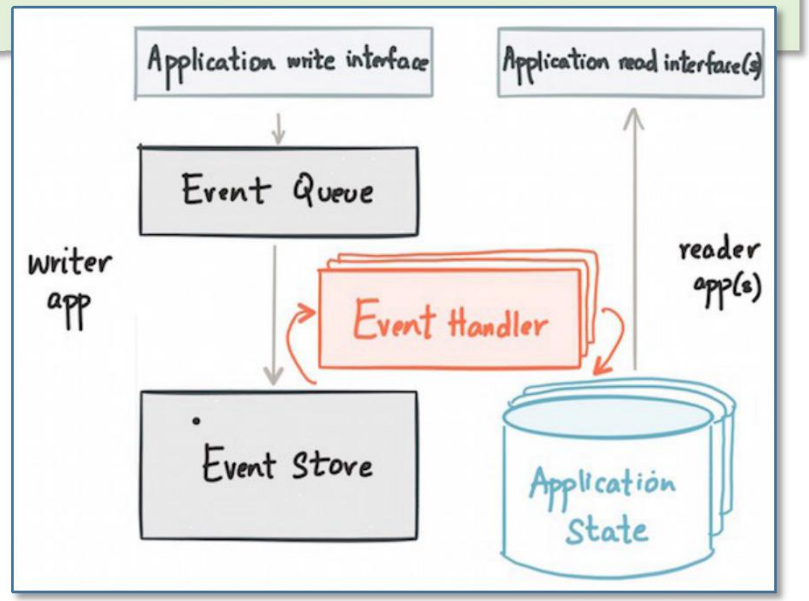
- **Implement** → Event sourcing design pattern
- **Design** → Event driven architecture
- **Do** → reactive processing

# Event Sourcing | How it works?

**What is event sourcing, then?**

- A software design that
  - → captures events (in correct order) as they happens in the system
  - → store these events in a transactional database
  - → never persists/stores current states (determined from events)
  - → always use the stored events to determine state

- The design is compose of
  - ➢ models for events and states, a event store, and reactors
    - Where event store is a composition of msg queue, and a persistent store to save events, and reactors are the event handler that either generates more events or creates data views.

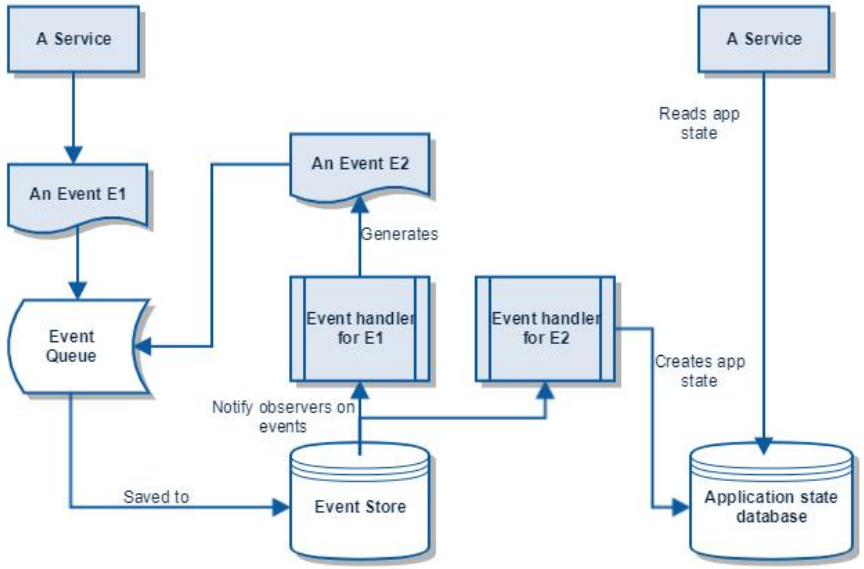


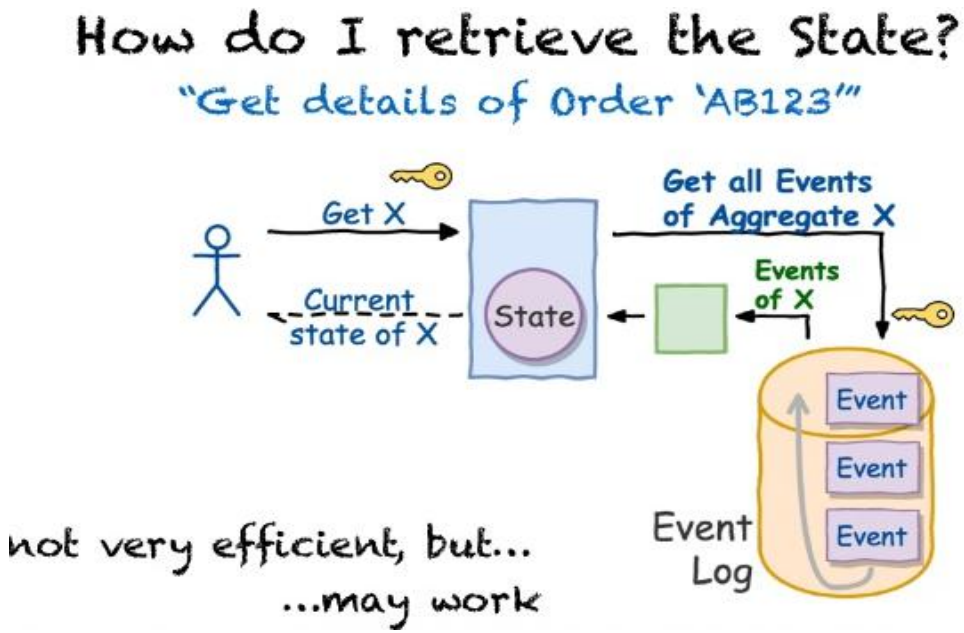

**How event sourcing works?**

- The events are captured on a medium that ensures ordering; as multiple services can generate them in any order.
- The events are then stored in a storage that provides transactional reads and writes.
- Registered observers, the reactors, are engaged if their event-of-interest is being recorded.
- The reactors also known as event handlers, then do processing, and can generate more events, and/or generate information that is then written to other databases.
- There can be multiple reactors for one or various events depending upon the need.
- The execution of event handlers is done asynchronously but in order; means handler of event E1 will be executed before the handler for event E2 because E1 happens before E2.
- At any given time, when any of the service or its module needs to know the current state, it get all the events from the event store, computes the state from it.
- The current state can be saved for some time period to overcome state computation cost.

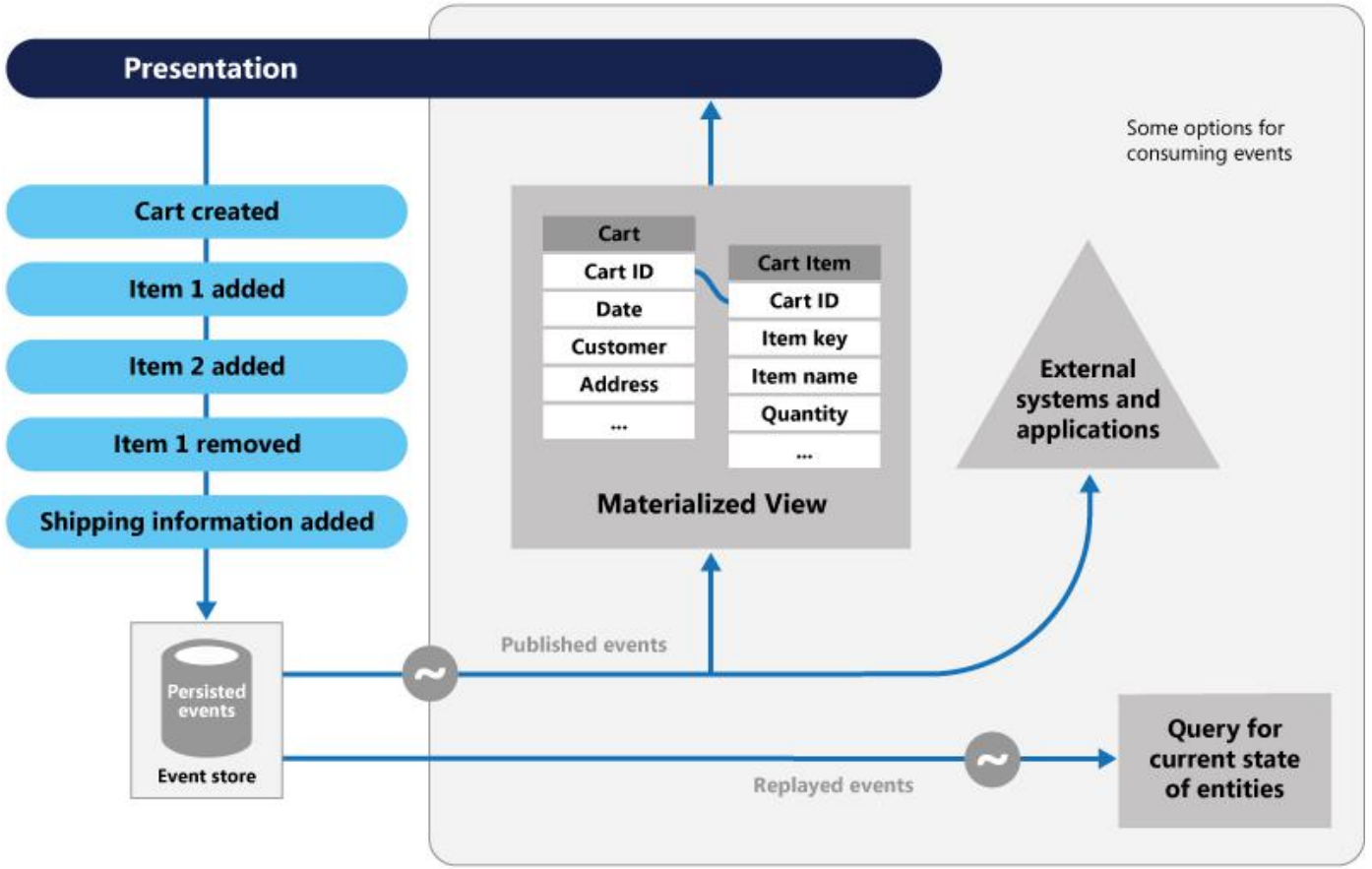
https://cdn-images-1.medium.com/max/1200/1*oSOTEY5OOcyjFvJ4utjiYw.jpeg

# Event Sourcing | Examples & illustrations …




https://docs.microsoft.com/en-us/azure/architecture/patterns/_images/event-sourcing-overview.png




https://image.slidesharecdn.com/introtoesandcqrs1-161125111615/95/a-visual-introduction-to-event-sourcing-and-cqrs-by-lorenzo-nicora-16-638.jpg?cb=1480603502

# Event Sourcing | Benefits …

- Complete rebuild ..
  - Complete resilience on establishing the state after failover
- Temporal query ..
  - Ability to analyze state and the path to it
- Event replay ..
  - Ability to simulate a state
- Consistent transactions ..
  - Efficient distributed transaction implementation
- Prevents complex synchronization between microservices ..
  - Paving the way for asynchronous non-blocking operations between microservices

**Implementations …**

- **Eventuate.io**
  - https://eventuate.io/
  - A framework ... with free and paid versions
  - Provides client library, and a server (with free version)
    - The server is a event store based on MySQL database and Kafka
  - With paid version, a Saas backend is provided
- Custom implementation
  - In spring boot microservice ... use
    - Reactor framework for event driven programming
    - Kafka for capturing events in order
    - MySQL, Redis, Cassandra to store events

# CQRS | What is it?

Is an abbreviation of **Command Query Responsibility Segregation**

## Why?

Sometimes in a systems, a high volume of reads and writes to a certain business entity creates service bottlenecks that not only hurts the reads and writes themselves, but also degrades other operations. In such situations, one solution is to employ CQRS design pattern for data access.

CQRS resolves the data accessibility issues at high rates but with a cost on added complexity, and relaxed data consistency.

Means ...

### Write requests as "commands"

- These commands are executed asynchronously
- A command is usually queued for later processing
- A command handler is called when the command is processed from the queue
- A command handler does the final business logic processing, and then writes/updates the data to the data store
- Upon successful write/update operation, an update event is being created for the query database to get updated

### Read requests as "query"

- Queries are designed and implemented to be fast
- The queries are usually being executed synchronously
- The queries based on the requested data, fetches it from the dedecated data store
- These data stores (from which queries get data) are being updated by the several events triggered by the command handlers
- Mostly queries do not prepare information, they simply get prebuild from their data stores

### Both writes and reads MUST have separate execution flows, end to separate persistent stores

- Both write and read will be implemented as separate services
- Both write and read flows will use separate models
- The schema for their data stores will be different, more optimized with model structure
- There won't be a single service and DAO layer for write and read flows
- Both write and read flows can execute in parallel, independently without each other knowledge
- It can use event sourcing implementation for tracking app/process/data state

# CQRS | In comparision with traditional data access strategy …

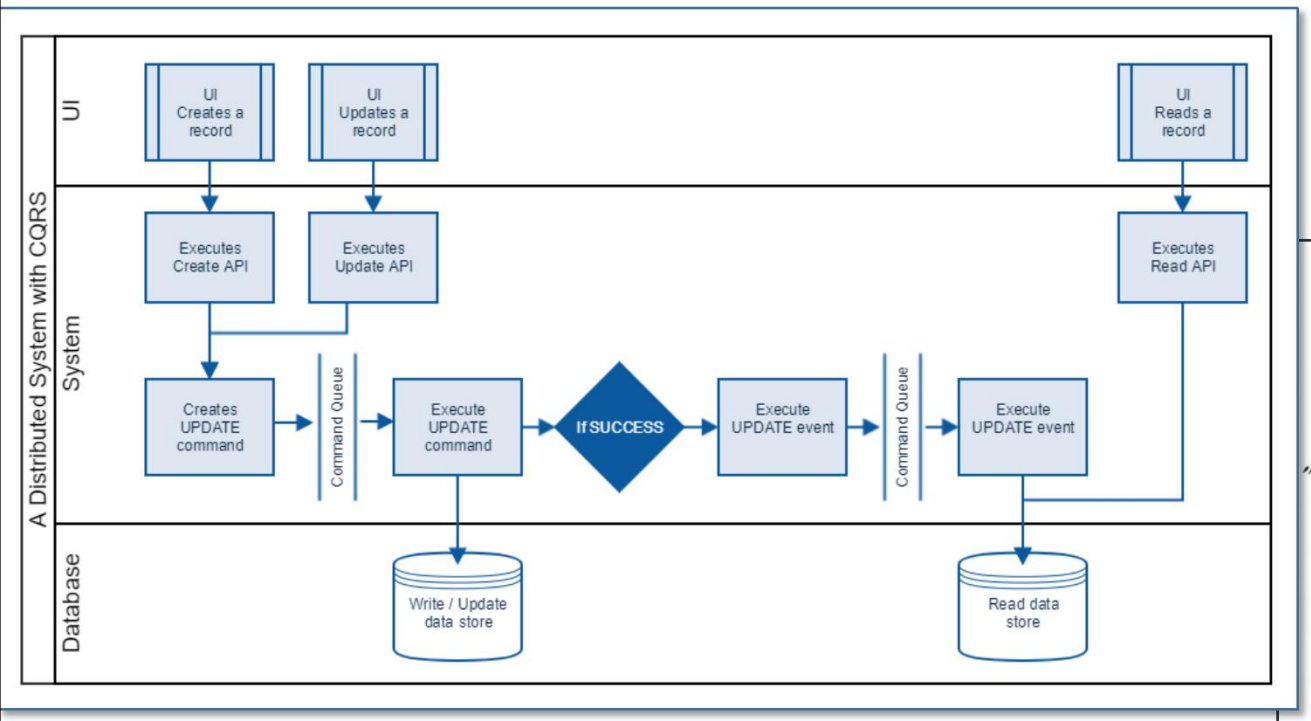
Difference in service layer with and without CQRS…

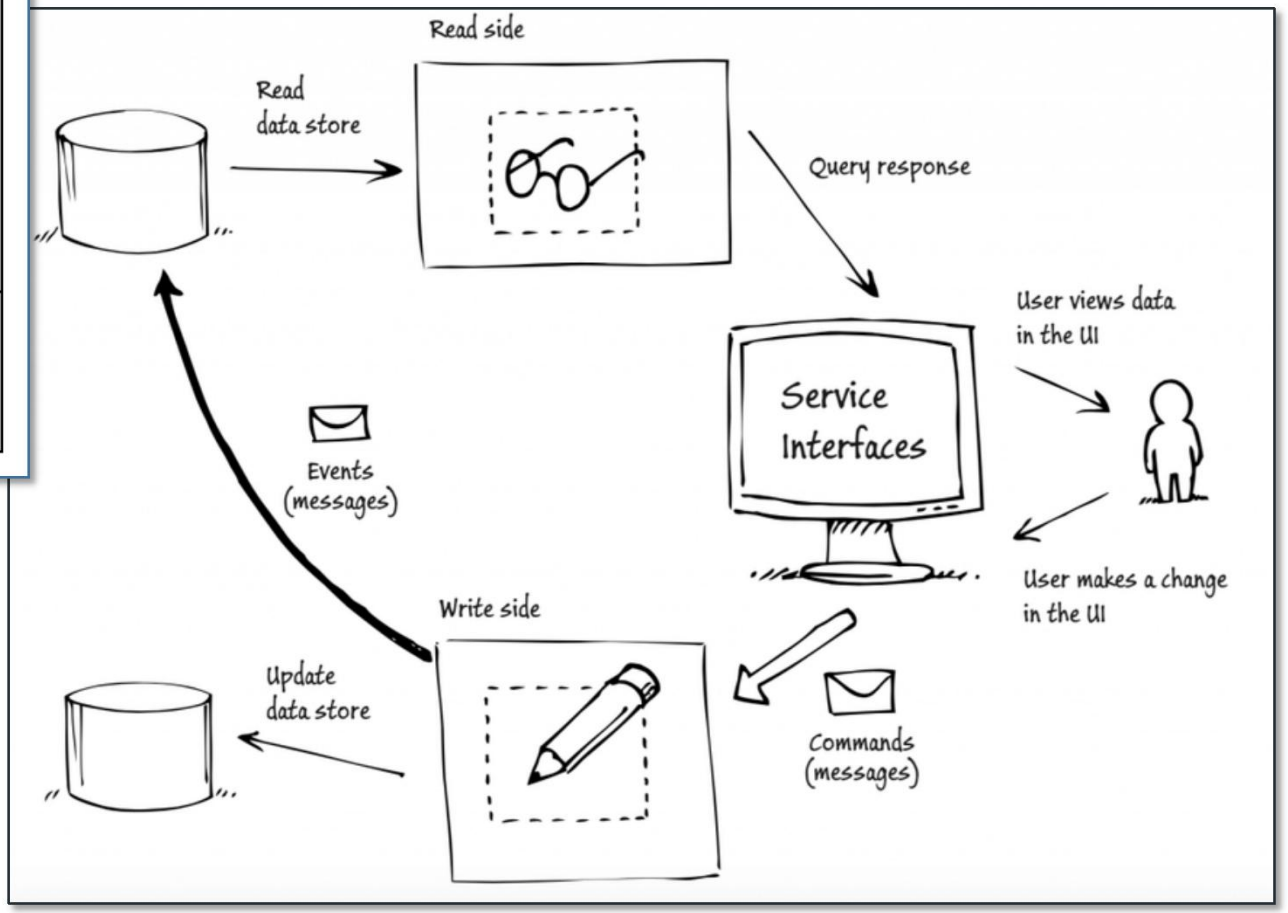
| With CRUD … | With CQRS … |
| --- | --- |
| • Single persistence store for all CRUD ops<br>• Single DAO layer<br>• A dedicated service layer<br>• Single model<br>• Calls are usually synchronous<br>• Data across reads and writes is usually eventually consistent | • Separate databases (or tables) for read and write ops<br>• Dual DAO layer; one for write and other for read<br>• No particular service layer<br>• Several models based on utility (usage); may have multiple for read based on UI<br>• Writes are async, whereas reads may be sync (usually)<br>• Explicit impl. is done to keep data consistent between the two data stores |

**CQRS implementation …**

- allows enormous data access scalability by detaching the write from read data flows, and making all writes asynchronous.
- allows high resilience to failures by preventing deadlock and starvation scenarios within the processes on data access.
- promotes least coupling between various read and writes data flows, hence enable easy management and maintenance.

# CQRS | Examples & illustrations …




https://blog.eleven-labs.com/assets/2015-04-07-cqrs-pattern/cqrs_pattern.png

# CQRS | Some facts …

- It increases the system complexity many folds.
  - Hence not recommended, and only use when there is no other solution … and for those business entities that makes the difference.
- Both command and query data stores can be scaled independent of each other.
- Data validation must be done before submitting the command so that user can be notified immediately.
- Business logic validation may fail later on, and hence there has to be a user feedback mechanism to notify failures.
- Since commands are being executed asynchronously, many can be execute out of order, and hence client must be design with this fact.
- Commands execution can be suspended due to unavailability of certain other services such as the database, etc, and hence can run later when the required downstream system gets available.
- A business entity write can be handle by multiple commands with different models as each has its own unique source, and provide data with certain variation.
- A single command handler or a collection of similar ones can be implemented as a separate microservice.
- Since the command database is not used for queries, data can be stored as a single object/document.
- Since reads are on different data store, they all can be made in parallel without locks; whereas writes thru commands can be serialize on their own data store.

**Implementations … Axon framework**

- https://axoniq.io/
- A framework … with Spring boot support
- Provides framework the establish abstractions for commands, events, and queries
- Provides support to implement event store based on JPA, JDBC, or MongoDB
- Can also be used to implement event sourcing

# Communication Concerns ...

The most complex challenge in realizing microservice architecture is not building the services themselves, but the **communication** between services. As this communication that defines the limits for the **scalability**, **resilience** and **efficiency** in the overall system.

# Communication | What are the concerns?

Communication ... is one of the core aspect of any distributed system.

- A good communication design leads to an lean and efficienct microservice architecture.
- A good communication design requires ...
  - parties to use efficient protocols,
  - lean dialogue structures and
  - use of best fit communication pattern

In core sense, communication can be either

- **Synchronous** ... Blocking call
  - means, the caller will going to get blocked until the callee return him the actual result
- **Asynchronous** ... Unblocking call
  - means, the caller will immediately get return and get his control back for other stuff, whereas the callee return the result later when it gets ready.

**Some common concerns ...**

| | |
|---|---|
| **Communication protocol** → | Must be efficient enough with least bandwidth requirements and infrastructure dependencies. Should be capable enough to provide all the communication functions. |
| **Dialogue structure** → | The dialogues are usually hierarchical in structure. Having a dialogue with compact structure leads to more efficient use of bandwidth, and memory requirements. That is why JSON is more efficient then XML, whereas protocol-buffers are more efficient then JSON. Binary is more efficient than textual structures. |
| **Communication cost** → | How much resources are being engaged in a communication session. Bandwidth, memory size, CPU cycles, and number of network mediators are all resources. More resources lead to more cost. A communication with least cost is the best communication. |
| **Communication resilience** → | Means whether the communication is capable of being resilient. A communication form that can enable the parties to employ strategies like circuit breaker, and load balancing to implement resilience is the best form of communication. |
| **Security** → | Ability to implement adequate form of security depending upon the need. |
| **Easy interface** → | Means the communication interface is so simple and easy for the new systems to implement integration. |
| **Effective & efficient fault tolerance** → | Means the communication is flexible enough to enable communication parties to employ various fault tolerance strategies for effective and efficient resilience. |

# Communication | Types ...

A communication can be one of three types; **command**, **query**, and **event**.

A **query** is a similar to command, but it is being used to retrive certain data without doing any alterations.

- Queries are quite simple and usually has less execution cost as compare to commands.
- Unlike commands, queries are usually implemented as synchronous calls.

An **event** refers to an occurrence in the past. That simple statement carries a number of consequences:

- Events are immutable.
- The emitter can be completely oblivious of the event listeners. Just publish it somewhere and let consumers react to it as they see fit.
- The listeners may also be oblivious of the emitter — in case the event messages flow through a shared channel (eg: a message queue), this is the only address they need.
- The overall process is async by nature;
- Therefore, this pattern allows fairly decoupled communication. The event emitter is saying “look, buddy, this is what happened, do what you will”.

A **command** happens when a client system needs an action to be performed by a given service provider. The client may also need a response (in either a sync or async way) informing about the operation result.

- Command promotes some coupling, as client (and potentially provider) are made aware of one another;
  - Coupling becomes tighter in case the client needs a response, as this means it will need to a) wait and b) handle all possible outcomes of the operation (including unavailability);
  - Coupling becomes tighter in case of synchronous calls, since the provider needs to process the request at the time it is made. For an in-between solution, a common pattern is to provide a simple “accepted” answer synchronously and do the heavy-lifting in an async fashion;
- Time-bound operations increase interaction complexity: if a client issues a command and needs an answer in up to a minute, what should it do after the timeout? What should it do if the provider still processed the request after, say, 2 minutes?
- Commands are usually being designed to process asynchronously so as to make the system efficient like in CQRS and ES design patterns.

# Communication | Strategies ...

There are few well defined strategies for communication between two software bodies. These strategies defines how a communication will gonna happen between the parties. These strategies are:

**1) Synchronous Calls ...**

The probably easiest communication pattern to implement is simply calling another service synchronously, usually via REST.

- Timeouts are one of the major concern in this strategy, and can also leads to inconsistencies. It creates strong coupling.
- Queries are mostly being implemented as synchronous calls. Whereas sometimes commands are also being implemented with this strategy.

**2) Simple Messaging ...**

The is the easiest asynchnous type of communication pattern where the sender submits its message and forgets. And on the other hand, the receiver or consumer gets the message later from a broker service.

- It provides implicit automatic retry, and hence enables guranteed delivery. Moreover it promotes loose coupling. However, there is a single point of failure which is the message broker.
- Commands are usually being implemented with messaging strategy.

**3) Transactional Messaging ...**

It is an special kind of messaging that reduces the need for 2-phase-commit in a transactional system. In this strategy, one implements a message journel (table) on both sending and receiving parties. When a message is being send, it is instead inserted to the journel as part of the transaction, and only being submitted to the queue when transaction got committed. And on the other side, when a message is being made available it is first add to the receiver's journel. When the commit is made, then the associated message in the journel is taken out and processed.

**4) Zero Payload Events ...**

This is the message queue used for domain events with data as the pointer to the actual payload in some other persistent store like a database.

- The pointer can be a URL to a resource whose sync call returns the data, or it can be an ID to a database the holds the actual payload data.

**5) Data Feeds ...**

It is a collection of data or events in strict chronological order as a single document, published or made available by a dedicated service. The consumer can then asynchronously read and process the events in the feed in his most convientent time or when there is a demand.

- It decouples the publisher from the consumers. However the consumer has to keep track of the time/event it has seen till now.
- Moreover it requires a separate persistence from where feed can be generated.

**6) Pub-Sub ...**

It is special kind of messaging, where a topic is being created. The sender “publishes” the data on the topic, and all the receivers as “subscriber” to the topic receives the notification of a new message.

- It is now-a-days one of the most implemented strategy. It prevents the subscribers from polling the new messages, and hence get the notification when new messages are available.
- All the subscribers on a topic gets the message. Hence multiple topics are being created for different types of messages.

**7) Message Bus ...**

It is a sophisticated software that does the orchestration, and integrate several applications, services, and microservices. It has all the intelligence and capability for doing message translation, transformation, and filtering.

# Communication | Implementations & Technologies ...

| | Technologies | Implementations (as frameworks, libraries, systems) |
|---|---|---|
| Synchronous Call | REST | Spring Boot Web, Jersey, Light-4j, Apache HTTPClient, Netflix Feign, HystrixFeign, ... |
| | SOAP | Apache CXF, ... |
| | RPI / RPC | gRPC, Apache Thrift, Apache Avro, ... |
| Messaging | JMS | ActiveMQ, RabbitMQ, Tibco EMS, ... |
| | Non-JMS | Kafka, ... |
| Feeds | Atom | Spring MVC, JAXB, ... |
| | RSS | Spring MVC, JAXB, ... |
| | JSONFeed | Spring MVC, ROME, Jackson, ... |
| Message Bus | | Apache Camel, Akka, Guava |











# Communication | Integration Patterns ...

In microservice architecture, services can be integrated in various ways. Following are the possible integration patterns:

## Integration Patterns ...

### 1) Point to point ...

- Services will make direct calls to the target services without any mediation software.
- In this type of integration, it is the responsibility of the caller to implement resilience using circuit breaker design pattern.
- Similarly, the callee service should setup load balancing and fault tolerance by employing reverse proxy, and having multiple instances.
- Here service discovery can be use to determine the callee service address before actually making a call.
- This type of integration gives the most efficient communication, and usually synchronous.

### 2) API (gateway) routing ...

- All the service are integrated indirectly through a router.
- When a service wants to call another service, the call is send to the router, and as per router own routing rules, the request is relayed to the actual target service.
- The routing can be defined on the target service names, or API call types.
- In order to prevent API gteway to become a SPF, multiple instance deployment is done with a load balancer.
- It makes the calling service more light wieght, as now they donot need to know the callee service address, and implement the resilience strategy.
- All is now being handled by the API gateway.

### 3) Message Queue (Message Brokers) ...

- Services are integrated through message queues.
- Hence when a service wants to call a service, it sends a message to its message queue, with the source service address/identifier.
- This makes the integration more efficient, as now the call will be executed asynchronously, and queues can gurantee delivery and can provide fault tolerance capabilities.
- With message queue, one can impl unicast and multicast communication.
- Moreover, advance channels can also be implemented such as pub-sub.
- However, this introduces a mediating component in the infrastructure.

### 4) Message Bus ...

- Services are integrated through a message bus.
- The bus, defines the messaging flow, and provides orchestration of the whole process.
- Like message queues, each and every services receives and send message from and to message bus.
- As per orchestration, services are called and their responses are then send to other services.
- The message translation, transformation and filteration is often done by the bus itself.

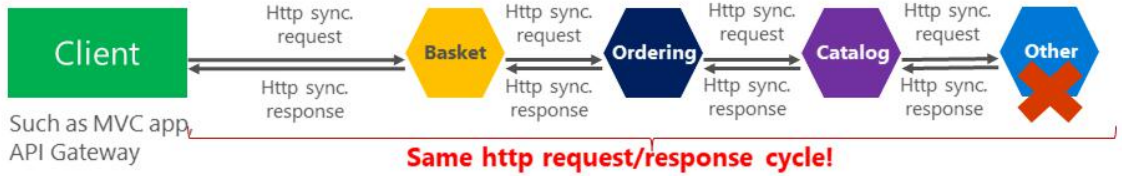
# Communication | Implementing queries …

There are 3 ways to implement queries; ❶ Making synchronous calls to the dependent services, and wait for the result, ❷ sending requests as messages with the reply-to address, and ❸ finally making sync call without waiting and then poll results periodically.

## Synchronous vs. async communication across microservices

❶ **Anti-pattern**

**Synchronous**
all request/response cycle

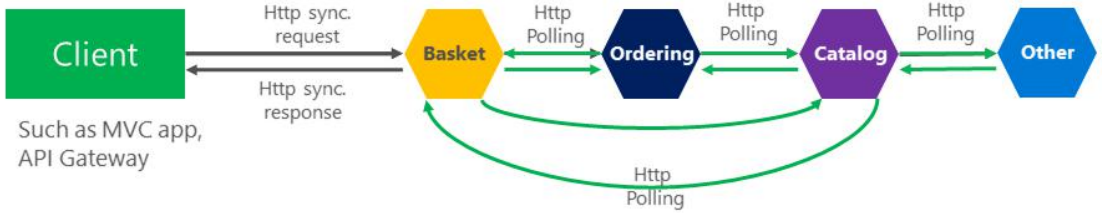



ALWAYS TRY TO AVOID THIS.

❷ **Asynchronous**
Comm. across internal microservices
(EventBus: like **AMQP**)

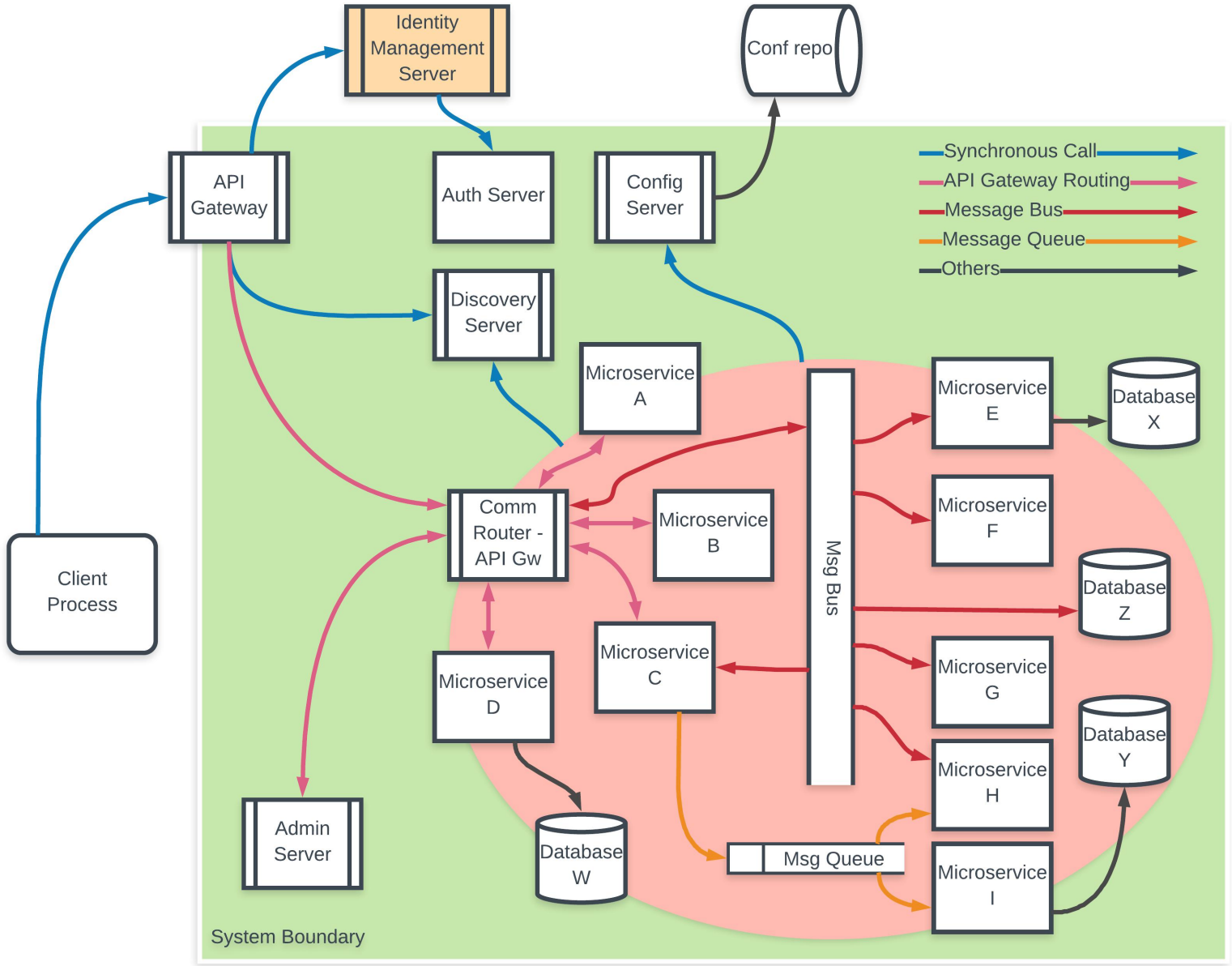

BEST WAY.

❸ **"Asynchronous"**
Comm. across internal microservices
(Polling: **Http**)




Although async, but worst way to get know the results; ie using polling.

# Communication | Example: Type, Strategy, Integration …




**Synchronous calls …**

- API gateway to discovery server
- All to discovery server
- All to config server
- API gateway to identity management server

**Messaging …**

- Microservice C to H and I

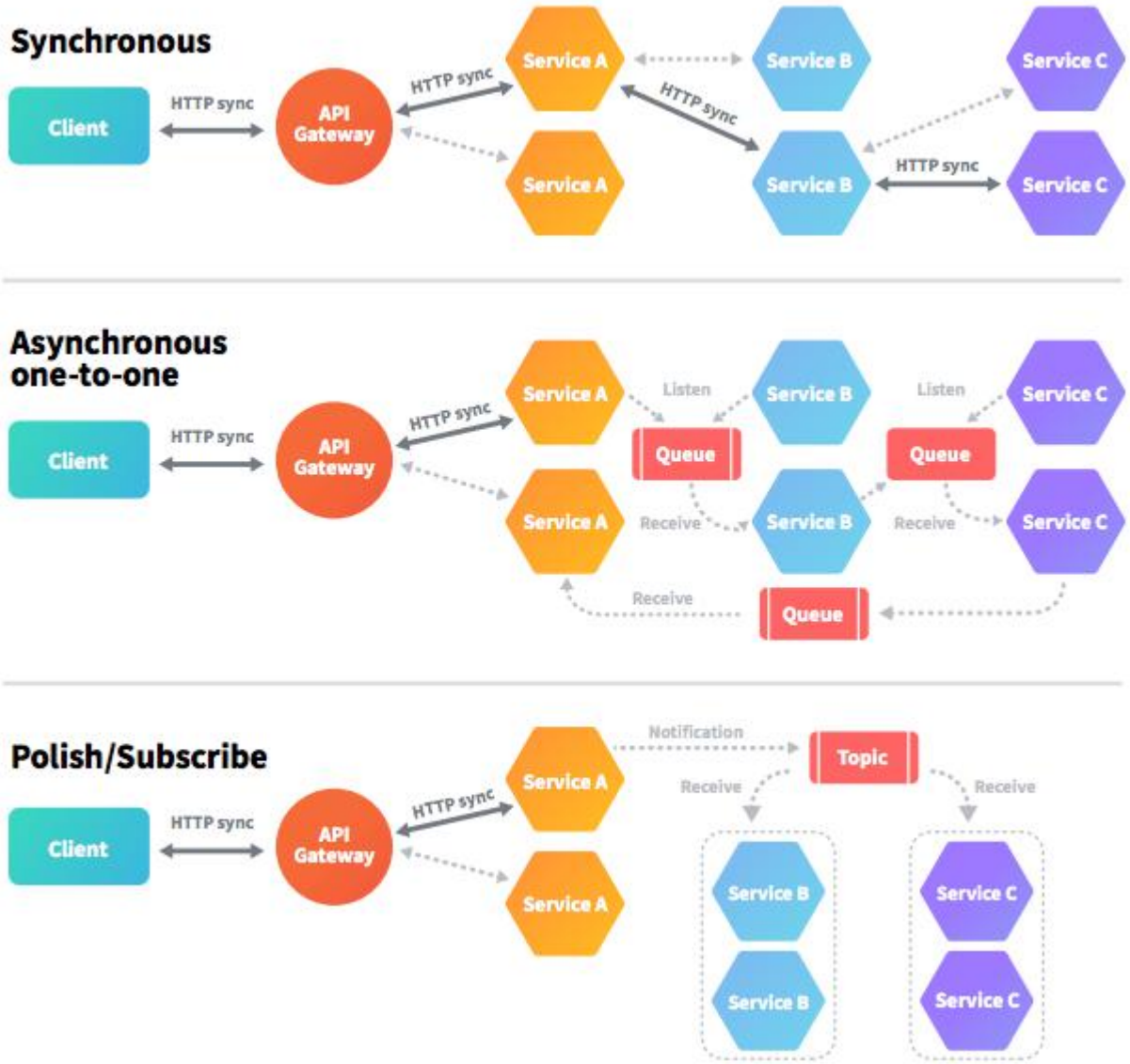
**Routing …**

- Microservices A, B, C, D, Msg Bus, API gateway

**Message/Event Bus …**

- Microservices E, F, G, H, database Z, with Comm router

# Communication | (cont.) Example: Type, Strategy, Integration …




In more complex architectures, there can be cases where those three communication types are mixed with each other. Then, some microservices are built on the basis of synchronous interaction, some on one-to-one messaging, and others on a publish/subscribe model.

# Communication | Dialogue formats

Message dialogues → communication constructs that are exchanged between the communicating parties.

As discussed before, these dialogue format is one of the core concern in efficient communication.

- A format that has least bandwidth requirements, gives the most efficient form of communication.
- However a format with more flexible structure gives more opportunities for enhancements, extensions, and complexity.
- The format can be custom build for a specific purpose, or a standard one can be used.
- A format structure define the dialogue reading processing requirements; for example, in case of XML, it requires a lot of processing. Whereas with compact formats like Thrift, it requires less processing.
- A format must be open enough to debug when needed.
- format vary along communication types, strategies, and integration patterns.

| Formats | Details |
|---|---|
| Header + payload | This is the oldest form of dialogues. Binary in nature. The dialogue always starts with a fixed size header that gives the header signature, size of the upcoming payload, and may be type of the payload. The payload is then read separately for a specific size already declared in its header. |
| XML | The dialogues are sturctured as XML documents. With that, a highly complex and nested data can be send. Moreover, there is no size limits. However, the structure itself is highly inefficient and wastes a lot of bandwidth. Is used with SOAP and RESTful APIs. |
| JSON | Another hierarchical format like XML, but requires very less characters to define a structure. It is native to JS language, easily processible in Java, and other popular high level languages, and one of the top choices in most of the RESTful APIs. |
| ProtoBuf | Introduce as a direct replacement for JSON. Provides compact format for more efficient bandwidth usage, and can easily be transformed from/to JSON. Is mostly being used in high volume APIs; e.g., APIs for consumer mobile apps, etc. It is being used in RESTful APIs and as independent structures such as in Machine learning frameworks like Tensorflow, Keras. https://developers.google.com/protocol-buffers/ |
| Apache Avro | Is data serialization/deserialization engine that can work with RPC framework. It provides a compact construct which is schema based, and gives least serialization footprint. https://avro.apache.org/ |
| Apache Thrift | It is a RPC/RPI framework that has its own data serialization format which is very much compact, and more efficient than JSON or protoBuf. However, it only works in synchronous calls (point to point integration), and requires clients to use communication interface. Usually used between services. http://thrift.apache.org/ |
| MsgPack | A direct replacement for JSON that can be use in any communication type, strategy or integration pattern. It gives better message compactness and serialization efficiency than JSON. https://msgpack.org/ |
| Fast Buffers | An advance and enhance replacement for ProtoBuf with more efficient serialization engine, and dialogue format. https://www.eprosima.com/index.php/products-all/eprosima-fast-buffers |
| FlatBuffers | An advance and enhance replacement for ProtoBuf with more efficient serialization engine, and dialogue format. https://google.github.io/flatbuffers/ |
| Cap'nProto | It is a RPC system with its own serialization engine. Very high throughtput, and limited to few languages like C++. https://capnproto.org/ |

# Communication | SOAP VS. REST

SOAP and REST are two API styles that approach the question of data transmission from a different point of view. SOAP is a standardized protocol that sends messages using other protocols such as HTTP and SMTP. The SOAP specifications are official web standards, maintained and developed by the World Wide Web Consortium (W3C). As opposed to SOAP, REST is not a protocol but an architectural style. The REST architecture lays down a set of guidelines you need to follow if you want to provide a RESTful web service, for example, stateless existence and the use of HTTP status codes.

| | SOAP (Simple Object Access Protocol) | REST (Representational State Transfer) |
|---|---|---|
| Design | Standardized protocol with pre-defined rules to follow. | Architectural style with loose guidelines and recommendations. |
| Approach | Function-driven (data available as services, e.g.: “getUser”) | Data-driven (data available as resources, e.g. “user”). |
| Statefulness | Stateless by default, but it’s possible to make a SOAP API stateful. | Stateless (no server-side sessions). |
| Caching | API calls cannot be cached. | API calls can be cached. |
| Security | WS-Security with SSL support. Built-in ACID compliance. | Supports HTTPS and SSL. |
| Performance | Requires more bandwidth and computing power. | Requires fewer resources. |
| Message format | Only XML. | Plain text, HTML, XML, JSON, YAML, and others. |
| Transfer protocol(s) | HTTP, SMTP, UDP, and others. | Only HTTP |
| Recommended for | Enterprise apps, high-security apps, distributed environment, financial services, payment gateways, telecommunication services. | Public APIs for web services, mobile services, social networks. |
| Advantages | High security, standardized, extensibility. | Scalability, better performance, browser-friendliness, flexibility. |
| Disadvantages | Poorer performance, more complexity, less flexibility. | Less security, not suitable for distributed environments. |

# Communication | RPC/RPI

**Remote procedure call or invocation** → a way to describe a mechanism that lets you call a procedure in another process and exchange data by message passing.

- It typically involves generating some method stubs for the client process that encapsulates all the communication details, and provides the client with the simple methods/procedure/functions to call. The stub implements the code for marshalling the request and send it to the server process, and the return procedure of getting the result.
- The server process then unmarshalls the request and invokes the desired method before repeating the process in reverse to get whatever the method returns back to the client.
- HTTP is sometimes used as the underlying protocol for message passing, but nothing about RPC is inherently bound to HTTP.
- Remote Method Invocation (RMI) is closely related to RPC, but it takes remote invocation a step further by making it object oriented and providing the capability to keep references to remote objects and invoke their methods.

**RPC over HTTP ...**

- RPC can also be implemented on top of HTTP protocol. Here will be seem like similar to REST, but technically it is very different.
- With this type of API, one is not concerns with the resources from domain. But rather, for each action, API is created.

| Technologies | |
|---|---|
| Google's gRPC | https://grpc.io/ |
| Apache's Thrift | http://thrift.apache.org/ |
| Apache's Avro | https://avro.apache.org/ |

RPC/RPI/RMI is one of the best communication strategy for fast paced internal communication.

- RPC always are synchronous calls.
  - However, one can make them asynchronous by returning immediately from the call; perhaps with a immediate result/response code.
- RPC are very efficient in communication, as they can be made on raw sockets.
- RPC gives a very great illusion at the client side for making a call. Hence making the development more easy and effective.
- RPC based on strict contracts.
- RPC calls can made context aware especially with RMI like frameworks.

**In comparison with REST interface ...**

- Same as RPC, REST are synchronous calls.
- REST is not efficient, and it is only due to its reliance on the HTTP application protocol.
  - Having an extra protocol layer, the HTTP, and secondly, HTTP is not design to be efficient for speedier communication.
- REST calls at the client code are very different from the usual method/function calls, and developer also has to provide the REST URI.
- REST calls can have loose contracts, or contracts with optional parameters.
- REST calls are always session less.

# Communication | Service contracts …

- When you have a business capability implemented as a service, one need to define and publish the service contract.
- In traditional monolithic applications one barely find such feature to define the business capabilities of an application.
  - In SOA/Web services world, WSDL is used to define the service contract. But as WSDL is insanely complex and tightly coupled to SOAP, it is not used for microservice contracts.

- Since microservices are built on top of REST architectural style, the same REST API definition techniques can be used to define the contract of the microservices.
  - Therefore microservices use the standard REST API definition languages such as Swagger and RAML to define the service contracts.

- For other microservice implementation which are not based on HTTP/REST such as Thrift, the protocol level 'Interface Definition Languages(IDL)' (e.g.: Thrift IDL) is used instead.

**Advantages of using service contracts …**

→ Enables design-first approach to API
→ Enables collaboration on API design
→ Allows easy integration
→ Generates client side code
→ Saves time and prevents errors when writing code
→ Assess and ensure quality
→ Generates effective and interactive API documents
→ Enable API marketing and publishing




Apache Thrift ™

# Communication | Best practices …

- Always tries to make **asynchronous calls** to other services.
  - For this, use **message queues**, **pub-sub channels**, or **message buses**.
  - Even for time costly queries, use message queue to send query request that also contains the **webhook** for the target service to submit the query results to the source.
    - Never ever use polling to query results for a previously made asynchronous request.

- If synchronous call is required to be made, use frameworks that implements **resilience patterns**.
  - Use **Netflix Hystrix** and **Fiegn HTTP** client for this purpose.
  - **Apache HTTPComponents** also implements some resilience patterns.

- Use **REST** for the APIs.

- Use **RPI/RPC** for implementing fast synchronous calls between services.
  - Use **gRPC** from Google, **Avro** or **Thrift** from Apache.

- For API calls, use **JSON** as dialogue formats. Use compact formats for APIs whose call volume is high.

- Always protect communication using **SSL/TLS** version 1.2 protocol whether it is external to the clients or internal with the other services. If possible make client certificate mandatory in SSL handshake.

- Always use **API gateway** to concentrate all external API calls from the client to a single node within the system; rather then exposing all services to the external clients.

- Always create **service contracts** using Swagger or any other platform.

# Implementing Security ...

A microservice architecture is designed with security as one of its core concern. The design along with adequate frameworks, tools, and tech enables a secure environment for the business workflows to execute across multiple microservices engaging multiple data sources.

# Security | Security breaches in microservice architecture …

Lack of security measures can lead to the following security breaches:

1. **Unauthorized access to certain services →** Means a hacker with a valid user session, able to made his way to a unauthorized service by forging his capability list, or exploiting vulnerabilities in the service itself.
2. **Congestion on certain services →** Means a hacker with a valid user session, using a DOS (denail of service) attack technique, able to create congestion on a single or multiple services which then either reduces the service efficiency, or make it completely unavailable.
3. **Failing certain services →** Means a hacker can trigger a vulnerability by crafting a certian request that generates exception on the service, failing its request processing thread that can lead to complete service failure which means failing other valid requests in the processing and queued states.
4. **Locking the services →** Means a hacker can trigger a vulnerability by crafting a certain request that creates a starvation or deadlock scenario either within two services, or with a database or external service. With that, the service althought running with least load, becomes non-responsive and unavailable to the new requests.
5. **Access to user data →** Means a hacker with a valid user session can query data and obtains its copy. The compromized data can of multiple users depending on the data-access-previleges to the victimized (hacked) user.
6. **Access to sensitive data →** Means a hacker hijacked a valid preveliged user session and/or able to hack a preveliged process by replacing it with its own, and then with preveliged access make queries to sensitive data, and make copies of it.
7. **Uploading files on unauthorized space →** Means hacker gaining access to a service environment either through hijacking a user session, or a previleged process, able to upload junk, or data files, that leads to shortage of disk space, which then can can then makes a service to cease operation as it requires space which is now not available.
8. **Installing backdoors or trojans in the system →** Means hacker gaining access to a service environment either through hijacking a user session, or a previleged process, able to upload its trojan horse, a backdoor, or a worm/virus software, and successfully installing and deploying it on the system for later execution. This can lead to very sophisticated and cordinated future attacks, and thus can do all sort of damages.
9. **Defacing of content and/or presentation →** Means hacker gaining access to a service environment either through hijacking a user session, or a previleged process, able to access presentation and/or content files, and replace them with his own.
10. **Others …**

# Security | Common causes of security breaches …

| | |
|---|---|
| **Weak passwords with no enforcement policy for periodic password renewal** | First there is no enforcement for stronger passwords, secondly there is no policy define to enforce user to renew their passwords once a month or a quarter. |
| **Weak cryptographic algorithms or cipher suits** | In SSL based communication or securing static data, use of weak algorithms or cipher suits can lead to data breaches when the hacker crack the encrypted data either through exploiting vulnerabilities or executing a exhuastive search. |
| **Vulnerable cryptographic libraries** | Use to old or vulnerable cryptographic libraries allows the hacker to crack encrypted data, gain access to the secure communication, or hijack a ongoing secure channel. |
| **Weak or no session validation** | Having no or weak validation for the session data in the incoming request usually lead to session hijacking scenarios, man in the middle attacks, request replay attacks, etc. |
| **Weak or no environment access policies** | Services environments with no or weak access policies often lead to environment breaches where the hacker get access to the environment, its filesystem, and other running processes. |
| **No proper segregation of resource files** | When resource files (which includes HTML, JS, CSS, images, configuration, and other files) are not properly segregated, means external from internal, readonly from modifiable, etc, then unauthorized access to one, can lead to other important files, leading more damage only because the files are not contained and protected from each other. |
| **No congestion control** | When there is no control on the incoming calls/requests then it can easily lead to network congestion scenarios which then makes services unavailable. With a proper control, services can still remain available by reverting pressure to the callers. |
| **No activity policing** | If the infrastructure has no intrusion detection system or any firewall, then it is impossible to stop an ongoing hacking attempt. Since no activities are being monitored, then a hacker can make damages up to any level, and he can go undetected with no one to stop him. |
| **Use of vulnerable libraries/frameworks** | Vulerable libraries are one the biggest culprit, and hackers use the exploits to crack these libraries and take control on the service that can lead to various type of breaches, and level of damages. Thus following adequate coding guidelines for security is not enough. One has to make sure that the libraries used by the service, and all the dependencies of these libraries are vulnerability free. |
| **Performing no or least frequent security scans** | Lack of security scans that include virus scans, malware scans, and audit trail checking leads to hacks that stays on the system for a longer time undetected, continously harming the system, its operations, and evidently the organization. |
| **Conducting no pen testing** | Penetration testing helps by unearthing issues, vulnerabilities, weaknesses, and flaws in the system that can be leverage by any hacker to make his attempt. When pen testing is not conducted after every version change or deployment, it leaves the system with unknown issues that are then unearthed by the hacker himself through various techniques, and then thus exploits these vulnerabilities for his advantage. |

# Security | (cont.) Common causes of security breaches ...

| | |
|---|---|
| **Bad system design ... Exposing internals** | When the internal services are addressible from the external clients/systems due to lack of consideration regarding having an edge service or a API gateway. This can expose the internal business operations, exposes entities, reveal vulnerabilities, and weaknesses, and hence makes the whole system more easy to hack. |
| **Bad system design ... Stateful services** | When the services are stateful, means they are holding information and data which is not persisted outside, can lead to business operation failures, system lockup, data loss, and other issues. |
| **Bad system design ... No load balancing** | When no load balancing is implemented, this can lead to system lockup as services becomes unavailable due to getting overloaded with work. |
| **Bad system design ... No resilence** | When resilience is not designed into the system, makes the system an easy target for a hacker. Such a system will cripple easily with lesser efforts leading to quick system failures which includes cascade failures, services unavailability, and congestions. |
| **Bad system design ... SPF** | When a certain service which provides an important function or used frequently is deployed solo without any backing instances. This makes a single point of failure, and hackers can bring down the whole system by simply targetting and failing the SPF services. |
| **Bad system design ... No protection of sensitive data** | When the credentials, pins, credit card numbers, social security numbers, etc are not stored encrypted or in a encrypted database or separately in secure vault, then data breach instances also leak this data also that bring more damage to the organization. |
| **Bad system design ... No security segregation of data** | When the data which is structured as per entity relations, but not structured, partitioned, contained as per sensitivity leads to sensitive data loss only because its other counterparts are being accessed. |
| **Bad implmentation ... No deadlock prevention and detection** | When in the implementation, resources are being accessed in various order in several concurrent workflows. Such access pattern leads to deadlocks that lockdown the overall service, and if detection is implemented, then the only way to get out is to at least fail one engaging workflow. Hence in the absence of detection, a service will keep running in the lockup state for ever. Databases also get into deadlocks but since they have detection mechanism, they manages to bail out from the situation, on the cost of failed queries. Hence due to bad implementation that causes deadlock, it always degrades a service performance, or make it unavailable, and becomes a tool for the hacker to bring chaos. |
| **Bad implementation ... Inadequate exception handling** | A service must have stronge handling of exceptions in its code. Exceptions can be raised from the code, and from the libraries, frameworks the serivce is using. Hence it is utter most important to handle these exceptions at proper code levels. Leaving an exception unattended can crash the service. Hence if a service fires up an unhandled exception on certian request or activity, then this becomes a tool for the hacker to exploit to bring down the service. |

# Security | (cont.) Common causes of security breaches …

| | |
|---|---|
| **Bad implementation … No logging or monitoring infrastructure** | When logging is not done adequately (means activities, operations, requests, and responses are not logged with full data and group IDs), can easily obscure a hacking attempt. Having no monitoring infrastructure can lead to situations when a system under fire goes unnoticed, and when it burns to the ashes, and services become unavailable then it comes to the attention. And with no logging, it becomes impossible to get know what and how it happends. |
| **Bad implementation …. No/weak request validation** | When request attribute/parameter values are not validated properly especially for boundary values, they lead to unknown execution scenarios that can lead to exceptions. Since these values are not expected, their effects are not being handled in the code which also includes exceptions. Each one of these attributes and their values that causes unexpected effects that leads to failures, or enables a hacker to take over the service process, becomes the vulnerability of the service. |
| **Bad implementation …. Direct use of request values without sanitization** | Services often get into issues when they use the request parameter values as it is without sanitization. This leads to various injection attacks including SQL query injection, HTML injection, etc. All of these attacks can do all sort of stuff. For instance, SQL query injection can do unauthorized updates, drops tables, reveal data, etc. |
| **Running services with high prevelige user context** | Due to lack of proper security context design, services are configured to run with previleged user context because they want to do some previleged user task or accessing some sensitive data. When these services are hacked, they can be turn into hacker's own tool that runs in the same previleged user context. With that context, the hacker tool then makes all kind of damages. |
| **No user authorization checks** | When there is no security role design for the user access, or the service has no implementation to verify the user role before granting him the access. This leads to unauthorize access to services on hijacked user sessions. |
| **Lenient session refresh/renew policies** | Session token expiration policy is define to enforce session refresh and renew process. This helps the system to get rid of hacked sessions. However if the policy is lenient, it makes defence weaker. |
| **Insecure internal communication** | With insecure communication between the internal services, or with the systems like database, msg queues, etc, enables the eavesdroppers to capture the communication packets, and then reconstruct the data in communication. Hackers when take control of a service, often do eavesdropping to get know data, access credentials to other services, and ongoing operations. Moreover, hackers install man-in-the-middle attacks where the internal communication gets mediated by a unauthorized entity. Such type of attacks bring damaages of various types. Hence internal services should do secure communication, for instance using SSL/TLS. SSL already authenticate the callee (the server) with the digital certificate. The caller (the client) should also be configured to authenticate using its own certficate. In that way, one can make sure that the communicating parties are guranteed that they are in communication with the right services. Moreover, a secure encrypted session will make it harder for the eavesdropper |
| **Keeping default signature** | Each service gives out a signature through which it can be recognized, and a signature can be so pecular that it can be use to determine the services version and/or known vulnerabilities. The signature is usually comprise of the interface port number, the underlying application protocol, its default banner, the comm dialogue construct, etc. Services configured to run with default signature can be hacked easily, as hacker can get know about its weakness through its usual signature. |
| **… and many others …** | … |

# Security | How to implement security ?

**System Design ...**
First step towards security implementation is right in the system design itself. Means the design must be done in way that enhances the security perimeter of the system, or at least donot weakens the security. Hence the architecture must not have any loopholes or weak spots that can be exploited later. Having a SPF is a big weakspot in a design. Aggregating and/or limiting communication paths, duplicating services, and having infrastructure that enables auto-discovery, load balancing, and auto healing leads to secure system design.

**Communication ...**
The communication between the services, and all the external ones especially with the client apps must be secured. For that, SSL or TLS with latest ver TLS 1.2 must be used. This will not only authenticate the communicating parties, but also make the whole conversation confidential and assure its integrity.

**Authentication ...**
All the communicating parties must authenticate others before doing any actual conversation. For that either SSL handshake can be used with X509.11 digital certificate with mandatory client side certificate. Or, employ some proper protocol such as OAuth2 that not only authenticates the client, but also provides a proper way to creating sessions.

**Authorization ...**
All the communicating parties must have well define authorizations and roles assigned through which one can be differentiated from others. These roles should be validated in each and every service so as to prevent unauthorized access that usually leads to hacks that can further leads to system sabotage and/or data theft.

**Accounting ...**
All the services engage in the system should do accounting for all their actions. This accounting is done through proper logging, assigning IDs to generate service traces, aggregating them for proper analysis and analytics. This accounting data helps to identify events, failures, issues, and trends in the system operations and thus help in creating the mitigation plans.

**Access Control ...**
All the services and their containers must have controlled access by restricting TCP ports to only service and its associates, and defining CLI access with only one or few known credentials or through VPN channel.

**Sensitive Data Identification ...**
It is essential to identify data and information that requires protection and is sensitive. Protection means to make the data confidential and limited to few users/services of certain authority; it can be to ensure data integirty; it can be to make data available all the time; or any combination of these three.

**Sensitive Data Persistence ...**
Data which is being ID as sensitive must be saved as per security requirements. For confidentiality, the data storage must provide encryption. For passwords, pins whose actual form is not required, they are transformed into hashs/msg-digest with their actual form discarded. Passwords are also being saved in specialized services called Vaults so that their access can be controlled and they can be manage separately.

**Pen Testing ...**
Penetration testing is a must practice for any system. It must be done at least once with every new version deployment. It helps identify issues in the system, and provides solutions to it also.

**Session Provisioning ...**
Session must be provisioned to associate access with context. Sessions should be generated and managed in a way that makes them least susceptible to attacks and hacks. Keycloak is one of the service that is being used to provision sessions.

**Session Validation ...**
Validation is required in each and every communication so as to minimize the chances of any hack. It can be done in one place if the design limits all the communication through it, or in every service otherwise.

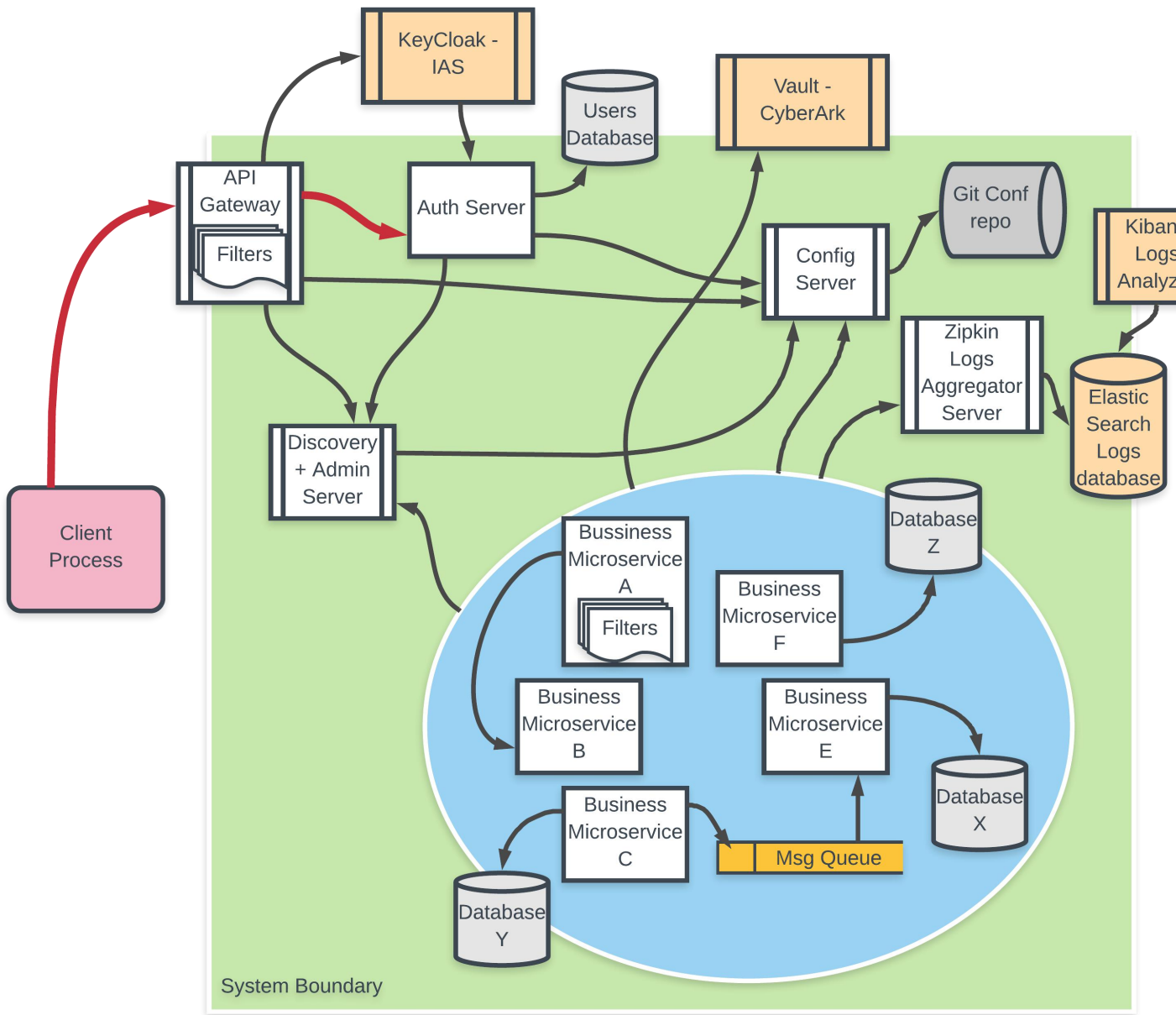
**Monitoring ...**
Each and every service must be monitored; as blackbox as well as white box. In the former case external tools are used to watch the services through their interfaces. In the latter case, service submit their activity logs with proper structure and/or emit health metrices periodically.

# Security | Technologies ...

| | |
|---|---|
| Load balancers | **Infrastructure**. They help in distributing load across multiple instances of a single service type. Some examples are NginX, Ribbon, Service discovery service (like Eureka, Zookeeper, or Consul), Docker Swarm, HAProxy, LoadMaster, etc. |
| Side-car proxies | **Infrastructure**. They are deployed along sideways to the actual microservices, and often works as forward proxies for making calls to other services. They provide several functions from dynamic routing, service discovery and registration, resilience patterns, gathering performance metrics, etc. They are often used with service mesh tech like Istio. Same examples are NginX, Envoy, Istio, Nomad, Finagle, etc. |
| Service discovery services | **Infrastructure**. They are being used as a naming services, and a load-balancer serivce within the microservice ecosystem. All the services register themselves to this service, and who ever want to send a reqest or message to another one, it simple resolves the target name to its actual address, and then make the call. For example Eureka, Zookeeper, or Consul, etc. |
| API gateways | **Infrastructure**. They are being used to converge and concentrate all the outside calls to the services from a single node. That single node then based on rules does the routing which also involves defining targe services, rewriting URLs, adding/removing HTTP headers, and doing general call filtration. For example, NginX, Zuul, Node.js, etc. |
| KeyCloak | **Infrastructure**. It is an Identity and Access Management system that provides authentication services for various types of identity servers/services. |
| Vault | **Storage**. A storage system for the sensitive data that includes passwords, pins, keys, and other information whose access is required to be managed per services bases. Spring Vault is the cliend side framework that provides the required integration with the Vault server. |
| Monitoring | **Infrastructure**. Each service must log its activities with proper grouping. Moreover, each service should also generate performance measures that can be analyze to determine failure conditions. Logging and monitoring is covered in detail in the later section. |
| Authorizat-ion | **Implementation**. Each service must do its own authorization check on each incoming requests, and only allow those that has the access preveliges to the access level. For that JAAS can be used. And Spring MVC filters can also be used here. |

| | |
|---|---|
| OAuth2 | **Communication**. It is a open protocol for authentication and authorization, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords. Spring and Spring Cloud have the security modules that provides this support. |
| JWT | **Communication**. It is a standard for representing session tokens. Abbreviation for JSON Web Tokens, provides a mechanism of communicating session tokens and other context material. Spring and Spring Cloud have the security modules that provides this support. |
| OpenID | **Communication**. OpenID is an open standard and decentralized authentication protocol for Internet Single Sign On; allowing users to be authenticated by co-operating sites (known as relying parties, or RP) using a third-party service, eliminating the need for webmasters to provide their own ad hoc login systems, and allowing users to log into multiple unrelated websites without having to have a separate identity and password for each. JAAS provides the support for OpenID services. |
| SAML 2.0 | **Communication**. Security Assertion Markup Language 2.0 is a standard for exchanging authentication and authorization data between security domains to implement web-based cross domain single sign-on (SSO). SAML 2.0 is an XML-based protocol that uses security tokens containing assertions to pass information about a principal between a Identity Provider, and a Service Provider. Java Spring Security provides the integration support. |
| AES \| RSA \| 3DES | **Algorithm**. These are the encryption algorithms that can be use to conseal a data. AES, 3DES are the one-key symmetric algorithms, whereas RSAS is a asymmetric algorithm. Java provides out of the box support as part of its core framework. These algorithms are used to secure sensitive data that may use again in the plain form. |
| SHA \| DSS | **Algorithm**. These are the one-way functions that produces message digest or hashes of fixed sizes. These are used to transform passwords and pins into a fixed sized text that cannot be change back to its original form. Java provides out of the box support as part of its core framework. |
| SSL \| TLS | **Communication**. These are the standard protocols use to secure a communication over application layer. It can be used over TCP/IP sockets or on top of several application layer protocols like HTTP. Its latest protocol TLS 1.2 must be used. |
| X509.11 Certificate | **Communication**. It is a format standard for digital certificates. It is being used widely in several security protocol whether identities assurities are made such as in authentication processes. |

# Security | Implemention example ...




**External Secure Communication ...** →
This is the secure communication using OAuth2 protocol and JWT based token exchange for session management, along with SSL/TLS protocol to secure comm.

**Internal Secure Communication ...** →
This is the secure communication using SSL/TLS protocol to secure comm. Here client certificates can also be enabled to authenticate the clients services also.

**API Gateway ...** Converge all calls to a single node, and then reroute the calls to their targets. Protects the services for direct attacks. Here it will also be enagage in the OAuth2 protocol, and will be the bearer between external and internal comm.

**Securing Session ID + Adding Session Context ...** This is done in API Gateway by its Filters. Hence all the calls routed to the internal services donot contains the OAuth2 tokens. However they contains all the session context info that is required for their operations.

**Session Validation ...** It is implemented as Filters in the API Gateway. They validate the token and session info within the JWT envelope in the incoming calls. Upon verification, the call is further taken for route processing.

**Session Generation ..** It will be done by the KeyCloak IAS service. It can further integrate with LDAP, identity servers, and social networks for certain users. It can also integrate with a local Authentication (Auth) server to authenticate for local users.

**Client Side Load Balancing + Service Discovery ...** This is being done by the Discovery server. All the services infra and business register themselves with the discovery server against a name. And during making a call, the discovery server is contacted first to determine the target address.

**Secure Configuration ...** All the services (both infra and business) get their configuration from the Config Server, that uses a secured repository as a persistent storage. The repo is kept private, access with secure keys, and over TLS protocols.

**Secured Sensitive Info/Data ...** All the credentials, pins, keys, and other sensitive data is being saved in the Vault server e.g. CyberArk. The Vault API is then being used by the services to effortlessly get the sensitive info without doing explicit operations.

**Service Monitoring ...** All the business services send their performance measures to the Admin service embedded with the discovery server.

**Access Authorization ...** Is done by all the business services, and is implemented as Filters as shown in service A but implemented in all business services. These filters from the session context determine whether the user can access this service.

# Security | Miscellaneous ...

## Java & Spring Tech ...

| | |
|---|---|
| **Java Crypto Libraries** | Java core libraries contains complete support for the crypto algorithms that can be use to encrypt decrypt text, generate hashes/message-digest, creating digital signatures, processing and generating digital certificates, and doing SSL/TLS communication. |
| **Java Spring Security** | Java Spring Security is the library that provides support for all authentication type protocols that includes OAuth2, JWT, OpenID, SAML 2.0, etc. |
| **JAAS** | Java Authentication and Authorization Service is the Java implementation of the standard Pluggable Authentication Module (PAM) information security framework. JAAS has as its main goal the separation of concerns of user authentication so that they may be managed independently. |
| **Java Spring Cloud** | Java Spring Cloud has dedicated modules for NSS OpenFiegn, Eureka, Zuul, KeyCloak and Vault support. |
| **Java Spring Cloud** | Java Spring Cloud has modules for actuators and admin that can be use to setup service monitoring. |
| **SLF4J / Java Spring Cloud** | Logging library that can be configured to push logs to external systems, databases, online services. Moreover Java Spring Cloud provides modules slueth and Zipkin that can be use to implememt service tracing. |
| **Java Spring Web** | Java Spring Web provides the filters construct through which one can define code that can run before and after the HTTP request. Similarly, Java Spring Security provides a filter chain that can be use to define authorization policies for a service. |

## Enabling CORS ...

- Cross-Origin Resource Sharing (CORS) is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.
- Spring Security has the proper implementation of CORS, and it is disabled by default.
- In order to use CORS, either declare it on the resource controller, or configure it in the global CORS configuration.

## Enabling CSRF Protection ...

- CSRF stands for cross-site-request-forgery, and it is a type of attack that enables a hacker to forge a call to a service with a valid user context.
- Its protection is easily possible, and it should be enabled all the times.
- Spring Security has the proper implementation of CRSF protection, and it is enabled by default.
- One can still disabled it either from the code, or from the spring configuration.
- Having CSRF protection enable will require CSRF tags in the incoming requests. Hence it is important to know how to have this implemented in the client request calls.

## Session Management ...

- Session is being created to define a user access context.
- A session often comprises of tokens (one for access, and another one for refreshing access).
- Moreover it also contains the user ID, its role, its capabilities, and some meta data regarding session tokens itself.
- The session representation is being send back and forth between the user's client, and the service. This representation is often implemented with JWT.
- A microservice cannot persists this session within, rather it is saved outside on a database. Since this database is being access frequently on each user access, the database is often choosen as NoSQL database like Cassandra, Redis, or as a cache like MemCacheD.
- The session must always be verified every time when the access is made to the service; and for that full session data may be required that can be access from the storage.

# Security | Best practices …

- Always use **API gateway** to limit system access through one node.
- Always use **SSL** with REST APIs whether they are external or internal.
- Always use **TLS 1.2** as the version for the SSL.
- If possible, also use **client side authentication** in the SSL communication.
  - This will also authenticate the client to the server.
- Always build **authentication server** as a separate service.
- **OAuth2 with JWT** can be used in most of the applications.
- Always check **user authorization** and capabilities in every service so as to verify whether the call can be executed in this user context.
- Store sensitive information like system access credentials, user passwords, pins, keys and other secretes in **Vault** like systems.
- Use third party **identity access management systems** rather than creating your own.
- Make sure to keep your services environment up to date with the **latest security upgrades**.
  - Use automatic security updates.
- Use a **distributed firewall** with centralized control.
- Get your containers out of the public network.
- Use **security scanners** for your containers.
- **Monitor** everything with a tool.

# Designing Resilience ...

A microservice architecture most essential function besides being secure, is to be resilient in all the cases. Fault tolerance is more important than the ability to be scalable and efficient. Thus many businesses often use microservice architecture in their on-premise systems to have a highly resilient distirbuted system that can withstand failures.

# Resilience | Failures in distributed systems ...

The eight fallacies of a distributed system ...

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Belief in any of the above fallacies open up the system for one or more failure cases. Hence the system must be design with considering all the above cases.

**Failures are normal cases ...**
**but they are not predictable ...**
**means a system should be designed and implemented with functions that can act upon any time and can collaborate with others to prevent, detect, and recover.**

Failures in todays complex, distributed and interconnected systems are not the **exceptions**.

- They are the normal case
- They are not predictable

**Do not try to avoid failures. Embrace them.**

The functional requirements of a distributed system should also contains requirements to overcome failures.

→ For that, one has to have complete understanding of the type of possible failure scenarios, and how to deal with them.

→ None of the requirements should not based on assumptions (discussed in the left -- fallicies).

→ Has to define functions that keep the system alive even with dead/disabled modules.

→ Has to define functions that degrades gracefully in the worst case scenarios.

# Resilience | What it is ?

**Fault Tolerance ??** ... means the ability of an architecture to survive (tolerate) when an environment, or sibling services misbehaves by taking corrective actions, e.g, surviving a server crash or preventing a misbehaving API from bringing down the whole system.

**In a microservice architecture ... this means:**

→ Ability to make sure, the system works with severed functions
→ Ability to revive the failed functions
→ Ability to detect failures
→ Ability to detect events that leads to failures
→ Ability to make sure availability of functions

**Resilience → "Ability to handle unexpected situations, withstanding environmental changes, embrace load distribution and upon failure do quick failovers."**

**Approaches to Fault Tolerance ...**

1. Proactive
   - Predicts/infers failure through constant performance and activity monitoring; identifying events that usually leads to failures.
   - **Software Rejuvenation →** Purge services' processes/threads in question, and restarting them, failing all the requests processing in the process, and restarting them again.
   - **Self Healing →** Cornering culpirt resources, requests, and terminating them that leads the services' processes/threads to the original state.
2. Reactive
   - Detect failures quickly through availability monitoring, and then restores the system back to a sane state/condition.
   - **Checkpoint →** Saving system's state/database periodically, and upon failure restores the last checkpoint state.
   - **Restart →** Making microservices stateless so that when they fails, simply restart them, to deploy them again.
   - **Retry →** Upon failure, do retry of the call, message. Moreover, saving requests as they can be retried again if the previous one fails.

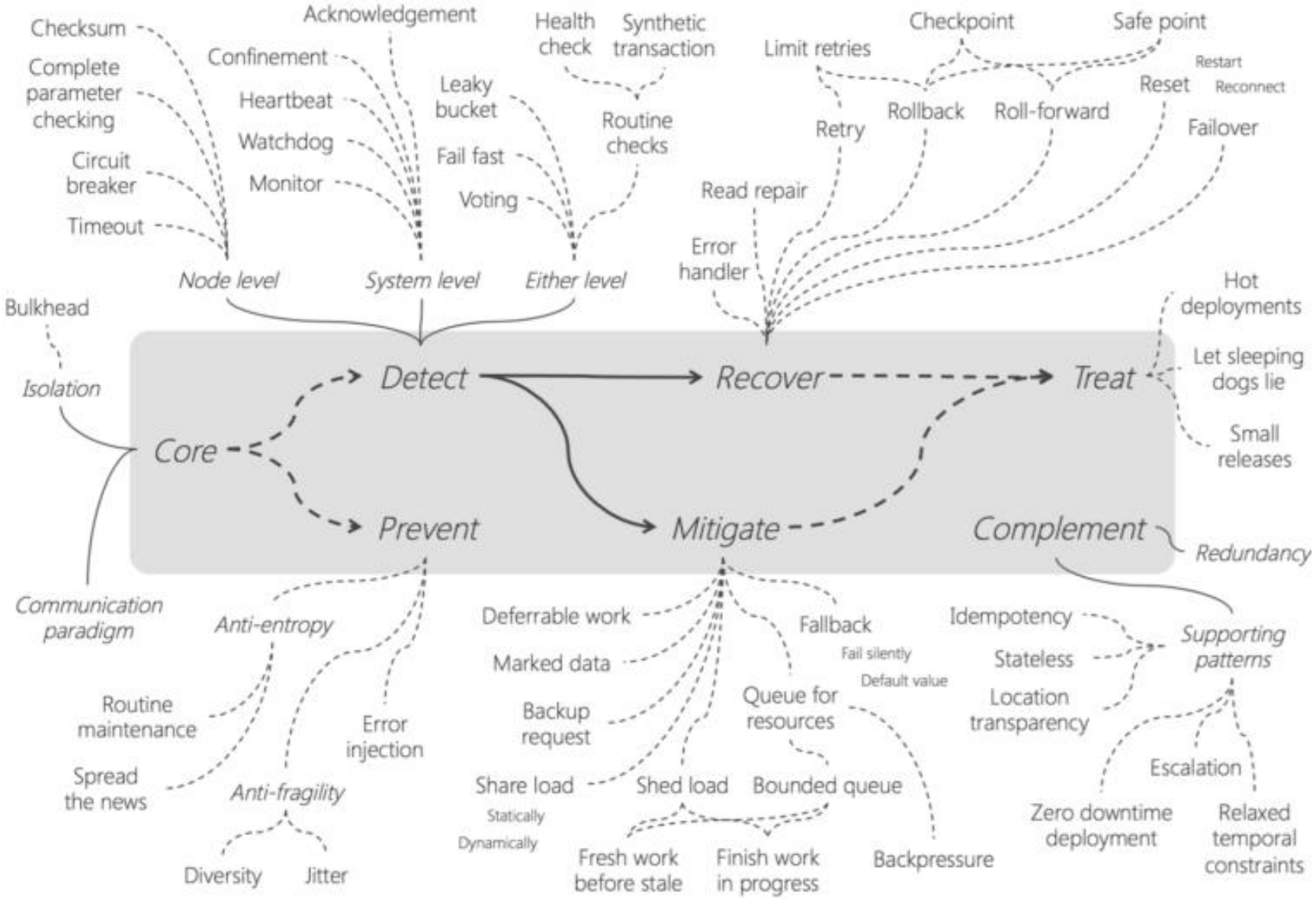
**Stability Patterns ...**

Stability patterns are used to promote resiliency in distributed systems, using what we know about the hot-spots of network behavior to protect our systems against failure.

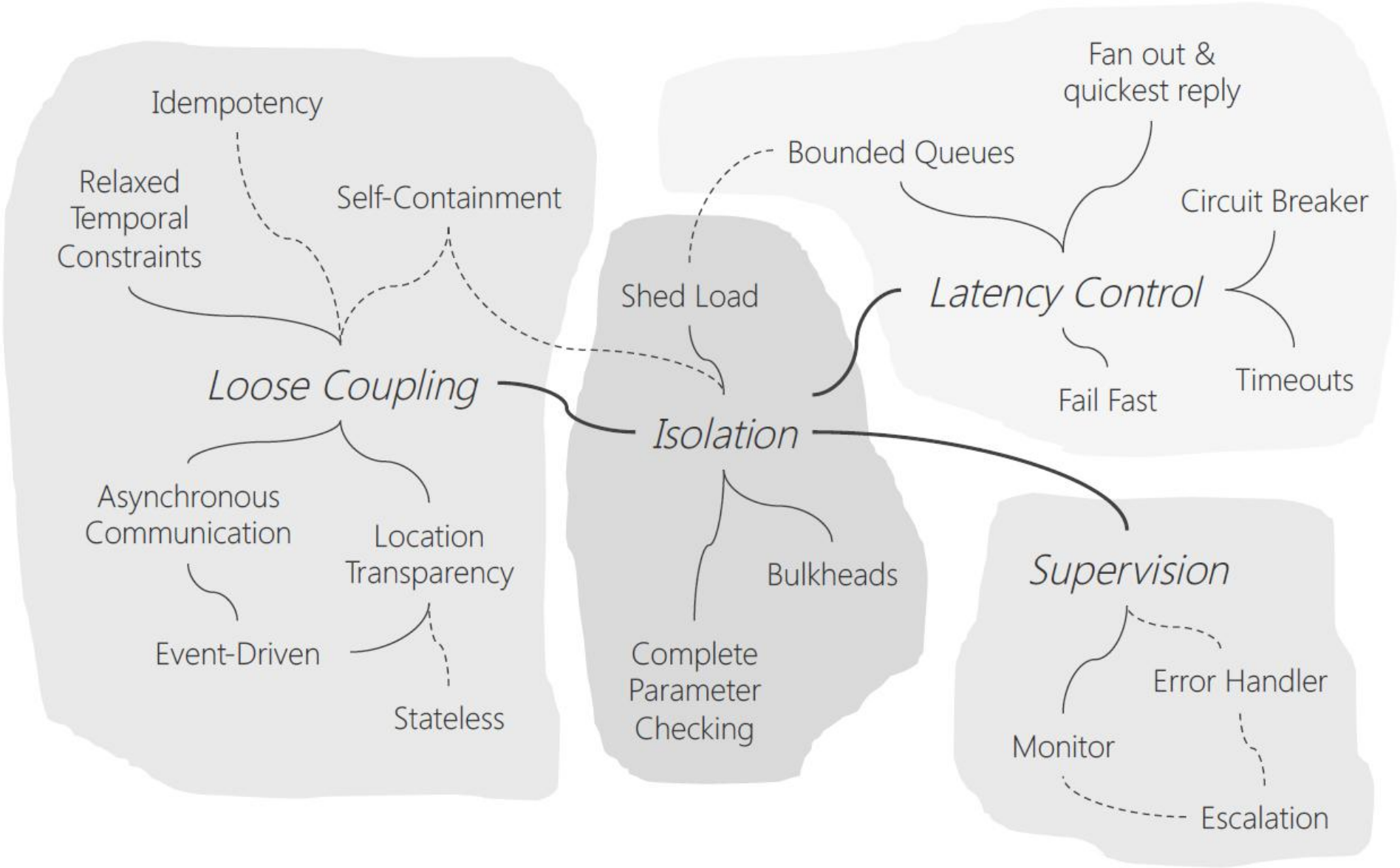
The following patterns are designed to protect distributed systems against common failures in network communication, where integration points such as sockets, remote procedure calls, and database calls (that is, remote calls that are hidden by the database driver) are the first risk to system stability. Using these patterns one can prevent a distributed system from shutting down just because some part of the system has failed.

A. Client side load balancing
B. Server side load balancing
C. Communication retry
D. Communication timeout
E. Circuit breaker pattern
F. Bulkheads pattern
G. Stress feedback
H. Service mesh
I. Others ...

# Resilience | Resilience patterns by Uwe Friedrichsen ...



# Resilience | Resilience patterns by Uwe Friedrichsen (small version) ...



# Resilience | Definitions …

| | | | |
|---|---|---|---|
| Isolation | Partition system such that it prevents cascade failures. | Latency control | Detecting and handling of non-timely responses; helps in avoiding temporal failures. |
| Parameter checking | Do request parameter syntatic and semmetic checking so as to detect broken/malicious calls. | Fail fast | Ability to fail immediately or at the earliest if it is known that the operation will gonna fail definitely, so that the caller can have the result (error, or detail response) at the earliest. |
| Loose coupling | Have least possible dependencies between the partitions/modules so that any failure can be isolated from affecting and impacting the caller module. | Fan out & quick reply | Send request to multiple workers, use quickest reply and discard all other responses to reduce probability of latent responses. |
| Asynchronous communication | Sender does not need to wait for receiver's response. | Bounded queues | Limit request queue sizes in front of highly utilized resources to avoid latency due to overloaded resources, and introduce pushbacks on the callers. |
| Location transparency | Sender does not need to know receiver's concrete location; useful to implement redundancy and failover transparently. | Shed load | Avoid becoming overloaded due to too many requests by having a gatekeeper in front of the resource (reverse proxy) that shed requests based on resource load. |
| Event driven | Instead of causing process on an emplicit synchronous call, make the process call implicit by binding it with an event; making the process execution more cost efficient. | Supervision | Constant monitoring to detect unit failures, provides failure handling and error escalation beyond the means of a single unit. |
| Stateless | Having no notion of state or data within the service runtime; hence state data is persisted and handled outside service context. | Monitor | Ability to observe unit behavior and interactions from the outside, and to automatically respond to detected failures. |
| Relaxed temporal constraints | Strict consistency requires tight coupling of the involved nodes, that compromises availability due to single failures; instead use a more relaxed consistency model to reduce coupling. | Error handler | Separate modules that are designed to handle error(s) properly, identifying steps for the failover process, as services themselves often only focus on business logic only. |
| Idempotency | Having idempotent operations, with deterministic results; hence limiting the environment impact on the results. | Escalation | Ability to escalate unhandled cases, exceptions, errors to designated services along with the contexts so that it can be processed properly and in failure scenarios relevent resources can be failover in a proper way. |
| Self containment | Having all the dependencies and required runtime bundled with the service artifact; thus having zero dependency on external resources. | | |

# Resilience | Load balancing ...

## Load Balancing ??

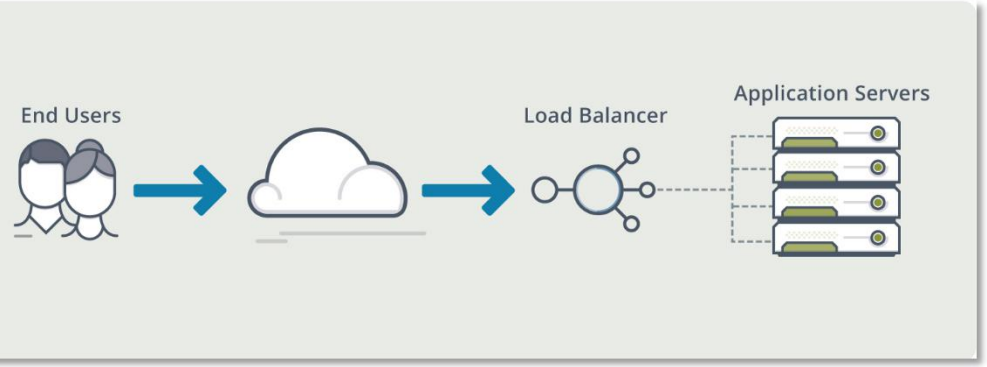
It is a fault tolerance strategy of having two or more service instances, available at the same level, distributes the overall incoming request load.

The load distribution is a function of a module that has the knowledge of all the available instances and their physical addresses, and hence all the requests are fan out on these instances based on some configured selection criteria.

The selection criteria can be based on:

- Round robin → Selecting the next in-line instance
- Least connections → Selecting instance with least connections
- Least response time → Selecting instance with best response time.

Load balancing can be implemented as **client side** and **server side** solutions. It depends on a service registry that has the knowledge for the services, and their active instances.




## Client Side Load Balancing ...

- The clide code addresses the target by its name, resolves the name to actual address using its access to the service registry service (or discovery server).
- Hence the service registry has the knowledge of all the actual addresses to the name, it always returns a next in-line address to the name.
- The cliend code, sometimes also caches the returned addresses to the name so that it can prevent calls to the registry service.
- Now the client code using any configured strategy selects the address for the name from its local cached list, and then do the call.
- If the call to the address gets fail, the address is removed, and next in-line address in the name list is used to make the call.
- Here the client can be an internal microservice calling another internal micrsoervice.
- Netflix Ribbon is one of the frameworks for implementing client side load balancing, and can be used to peer communication for synchronization, etc.
- Netflix OpenFiegn HTTP client has the implementation using Ribbon internally, and it can function with Netflix Eureka seamlessly.

## Server Side Load Balancing ...

- The client code unknown to the registry service, addresses the target service by its name.
- The client calls are proxied by a mediator service that aware of the registry service, determine the address to the name, and then rewrite the call with the actual address, and send it to the target service.
- As in client side load balancing, the proxy can do address caching, and can itself do the address selection based on the configured strategy.
- The main advantage here is the calling simplicity at the client side; as it is now only need to make a call, and the proxy will then do all the work.
- Here API Gateway also does the same function, but for the external clients.
- Internal services can use reverse proxy or side-car proxy to implement server side load balancing.
- NginX, HAProxy are some of the reverse proxy applications that be used to implement server side load balancing.
- Netflix's Zuul API gateway has the required implementation for the internal services, and can function with Netflix Eureka seemlessly.

# Resilience | Pooling, communication timeouts, retry ...

## Timeouts

It is a bailout strategy used in scenarios that requires activity acceptance from the other side. Without this bailout strategy, the caller might wait for ever until the other side responds.

In communication, timeouts are being used to define a time period that if the other side fails to responds within that period, the caller will simply returned an error (timeout error) so that it can then try other instance, or retry the communication again.

Essentially, in the low level sockets API defines two types of timeouts:

1. The connection timeout → denotes the maximum time elapsed before the connection is established or an error occurs.
2. The socket timeout → defines the maximum period of inactivity between two consecutive data packets arriving on the client side after a connection has been established.

It is one of the basic resilience support available in all the client libraries.

## Retry

It is another simple resilience support mostly available with timeout. It is a mechanism of retrying the communication again after the failure. In network communication, it is very possible to get the communication denied or failed on first attempt, and then accepted in the subsequent retry. It is often happened due to underlying network protocols like ARP, ICMP, DNS, etc that fails to perform their tasks in the first try, and succeed in the second or third try.

With retry, the client code simply keep a retry counter, and upon detecting failure in communication, repeat the communication work again till the counter get exhuasted. After counter exhuastion, the communication failure is returned to the client code. The counter prevent the client code from getting into infinite retries for a target service that is actually not available, or misbehaving.

Usually a wait time is being used between two subsequent retries and it is increased with a function like exponential so as to facilitate large time service failures, and unavailabilities. Like timeout, its implementation is available in most of the client libraries.

## Connection Pooling

It is a way of handling and managing openned connections. A pool is managed for a set of open connections. Initially the pool is empty, and for new connection request, a new one is being created, and once it is being used, is saved in the pool instead of closing it up. And thus on the second new connection request, the connection from the pool is served.

It is the most efficient way to managing connections. Moreover, with SSL based connections, the time taken on handshake is prevented on connections getting served from the pool.

Connection pooling works very well with HTTP protocol, and hence it is being used a lot for accessing HTTP APIs, HTTP based web services, RESTful APIs or RPC over HTTP, etc.

It is being implemented by most of the HTTP clients such as Apache HTTPClient, RESTEasy, OpenFeign, etc.

# Resilience | Circuit breaker pattern ...

Timeouts and retry has limitations →

- like the former prevents the clients from waiting forever on a unavailable service, and
- the latter gives resilience to communication on temporary/transient failures.

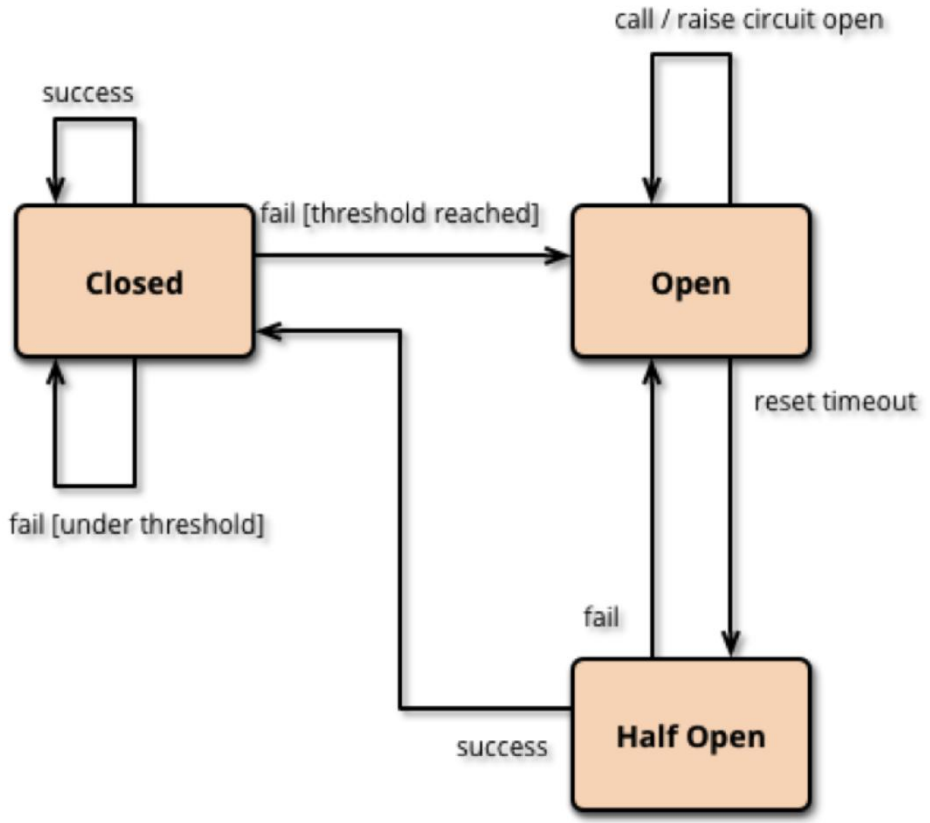
For resilience over long time failures, or perminent failures, circuit breaker pattern is being used. It is a resilience pattern for communications that prevents the client on unnecessary timeouts and retries for the target service which is simply not available indefinitely, or is perminently failed.

**What is circuit breaker ?**

Circuit breaker in software systems is very similar to the one in electric circuits. It wraps protected calls and makes sure they do not harm the system. Under normal circumstances a circuit is **closed** and remote calls are executed as usual. When a failure occurs, the circuit breaker takes note of it. If the failure rate exceeds certain threshold, the circuit breaker trips, **opens** the circuit and all further calls will fail immediately without reaching the remote destination. After some time a few requests are let through to the remote server (ie the circuit is **half-open**) to test if it is up again. If they succeed, the circuit is closed again, and all remote calls are executed as before.

Netflix's Hystrix provides the circuit breaker implementation. Moreover Netflix's OpenFeign HTTP client also gives the support using Hystrix.

The pattern can be implemented at either client (caller) or server (callee) side. In the former case the pattern gives resilience on the target service/host, and hence if it is not available, opens the circuit. In the latter case, the pattern gives resilience on the server operation; thus the circuit opens if the operation is keep failing beyond defined threshold.



# Resilience | Cache, fallback, handshaking (stress feedback) patterns …

| Cache | Fallback | Handshaking |
| --- | --- | --- |
| Cache can be used to impl resilience. With cache, pre-configured requests responses are made to cache within the service. When the request comes in the response is served from the cache instead of actually executing the request to get the response.<br><br>The cache response can be flagged dirty and refreshed with the corrected version. It is often handled by the service itself.<br><br>The cache can also be used within the fallback pattern. So instead of returning a default response as a fallback, cached response can be served that will be more appropriate. | With the fallback pattern, when a remote service call fails, rather than generating an exception, the service consumer executes an alternative code path and tries to carry out an action through another means. This usually involves looking for data from another data source or queueing the user's request for future processing. The user's call isn't shown an exception indicating a problem, but they may be notified that their request will be fulfilled later.<br><br>For instance, suppose you have an e-commerce site that monitors your user's behavior and tries to give them recommendations of what else they could buy. Typically, you might call a microservice to run an analysis of the user's past behavior and return a list of recommendations tailored to that specific user. If the preference service fails, your fallback might be to retrieve a more general list of preferences based off all user purchases. This data might come from a completely different service and data source. | An alternative is to detect an overload situation in advance. If an impending overload is detected, the client can be signaled to reduce requests. In the Handshaking pattern a server communicates with the client in order to control its own workload.<br><br>An approach to the Handshaking pattern is for the server to provide regular system health updates via a load balancer. The load balancer could use a health-check URI to decide to which server requests should be forwarded. For security reasons health-check pages are generally not provided for public internet access, so the scope of health checks is limited to company internal communication.<br><br>An alternative to this pull approach is to implement a server push approach by adding a flow-control header to the response. This enables the server to control its load on a per-client base. In this case the client must be identified. |

# Resilience | Bulkheads pattern …

Circuit breaker pattern implementation can protect and recover only from failures related to service interactions. To recover from other kinds of failures such as memory leaks, infinite loops, fork bombs or anything else that may lead to cascade failures and prevents a service from functioning as intended, one may need some means of failure detection, containment, and self-healing. Hence bulkhead pattern is another strategy that helps in this regard.

Bulkheads as used in their physical forms, are referred to resource partitioning in software design. Means breaking up the finite resources into groups such that, the processing in one group has no influence on the others. Hence resilience is gained from that fact that a failing process in one group do not impact the others in other groups. More importantly, this isolation also make sure that a process cannot possess more resources over certain define limit so that other processes can still function with the available resources in the system.

Through bulkheads, one can prevent service failures due to resource unavailability. E.g. Bulkheads can be configured like assigning 2 instances to a certain region. Now if that region bring a lot of load on the system, it will only affect the 2 instances, the others will keep on working servicing other regions.

**Bulkheads implementations …**

Clusters, instances, CPUs, threads, memory (heap), connections are the resources that are partitioned using bulkhead pattern.

Resource partitioning can be done as per caller that can be a *client*, an *application*, an *operation*, or an *end-point*.

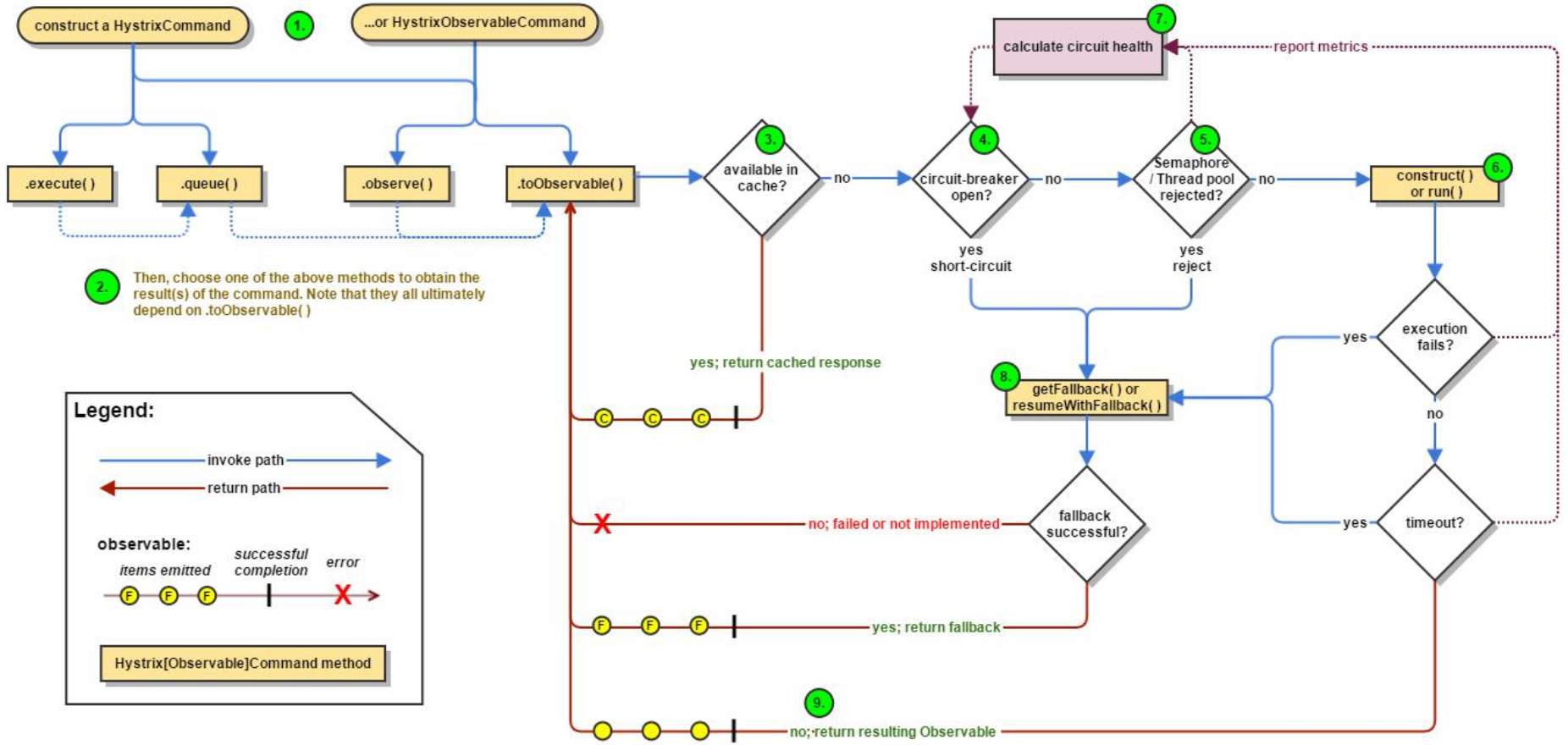
… means each partition is configured to service certain type or count of clients, applications, operations, or end-points. Hence through static pre-allocation of resources to the possible callers, one can make sure that no one can dominate others or influence others through starvations, or failures.

For cluster and instance type resources, usually request routing, and instance configuration is being used to create bulkheads. For CPUs, threads, memory, and connections software libraries are used. **Netflix Hystrix**, and **Resilience4J bulkheads** provides complete implementation of bulkheads pattern.

# Resilience | Hystrix

https://github.com/Netflix/Hystrix/wiki/How-it-Works

Hystrix is an implementation of several resilence patterns, which are **async-processing**, **cache**, **timeout**, **circuit-breaker**, **fallback** and **bulkheads**. It is part of Netflix OSS projects. The actual client code that does the communication can further implement **retry**, **connection-pooling** and **client-side-load balancing**.




## Hystrix execution workflow ...

1. The first step is to construct the hystrix command as a synchronous call, asynchronous call, or as a observable event.

2. There are four ways you can execute the command, by using one of the four methods of your Hystrix command object. These are (a) execute() — for synchronous call, (b) queue() — for async call, (c) observe() — for subscribing to the Observable that represents the response(s), and (d) toObservable() — for getting an Observable that, can be use to subscribe.

3. If request caching is enabled for this command, and if the response to the request is available in the cache, this cached response will be immediately returned in the form of an Observable.

4. Check circuit breaker state. If it is open take to (8) otherwise to (5).

5. Check for bulkhead (impl. for threads and semophores). If not available take to (8).

6. Here, Hystrix invokes the request to the dependency by means of the method you have written for this purpose.

7.Stats are being send back as feedback which is then used to maintain metrics that helps the circuit to open, or close.

8. Hystrix tried to revert to your fallback whenever a command execution fails. The fallback method provides a generic response, without any network dependency, from an in-memory cache or by means of other static logic.

9. If the Hystrix command succeeds, it will return the response or responses to the caller in the form of an Observable. Depending on how the command is invoked in step 2, above, this Observable may be transformed before it is returned to caller.

# Resilience | Sidecar pattern, & service mesh …

## Sidecar pattern …

Sidecar patterns provides a strategy that co-locates a cohesive set of tasks and functions from the parent/primary application into a separate module/service/application. This sidecar module although not part of the parent service, but works along side it, and offload certain functions and responsbilities. Its goes whereever the parent service goes. It shares the fate of its parent application. For each instance of the application, an instance of the sidecar is deployed and hosted alongside it.

Advantages of using a sidecar pattern include:

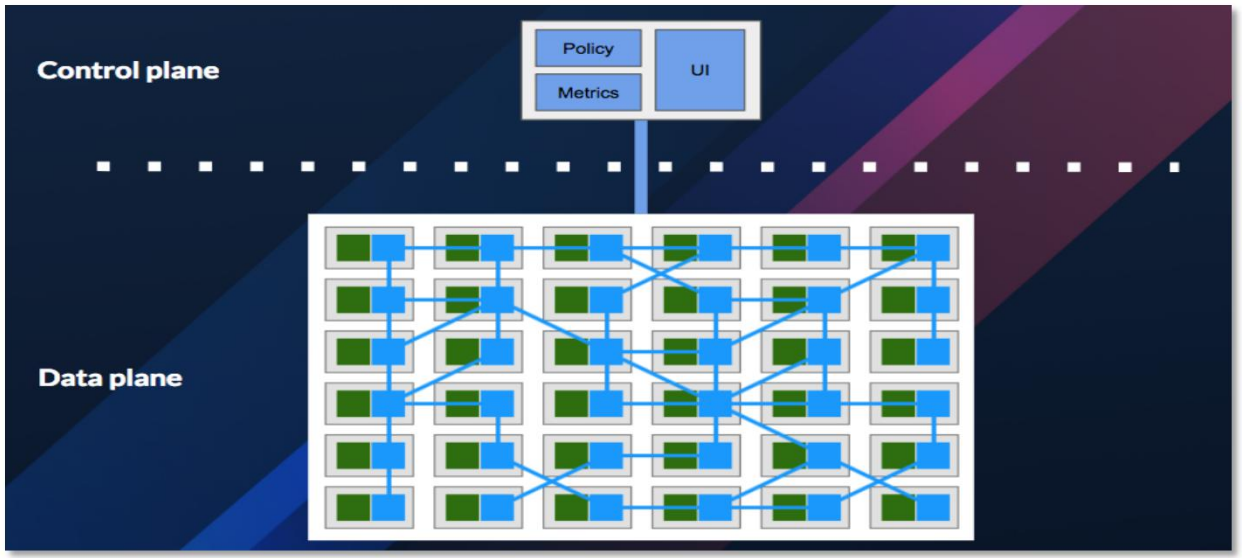
- A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language.
- The sidecar can access the same resources as the primary application. For example, a sidecar can monitor system resources used by both the sidecar and the primary application.
- Because of its proximity to the primary application, there's no significant latency when communicating between them.
- Even for applications that don't provide an extensibility mechanism, you can use a sidecar to extend functionality by attaching it as own process in the same host or sub-container as the primary application.

Sidecar modules usually offload those functions and responsbilities that are being used in most of the applications. Hence doing so, it enables the application developers to focus on application specific requirements, and simply offload other repeating concerns to the sidecar module.

**Problem:** In a microservice architecture, the services are designed to perform with max. resilience and participate in load balancing. Hence for that, each and every service than has to implement support for service registration and discovery, circuit breaker pattern with timeout, retry, fallback, and other patterns in its communication code. Hence a lot of code is being develop to implement load balancing, and resilient patterns. And the same repeating work is being done in each and every service.

**Solution:** Why not separate out the repeating impl. for the microservice patterns into a sidecar module, and use it along with the service which will now only holds the business logic.

When a microservice-architecture's communication infrastructure is comprised of sidecar proxies, each attached to a microservice is called Service Mesh. With service mesh, all the cross cutting concerns regarding secure resilient communication along with load balancing is being taken care by the sidecar proxies themself, leaving their services (parent processes) to only concern with the business requirements.



# Resilience | (cont.) Sidecar pattern, & service mesh ...

- A given microservice won't directly communicate with the other microservices.
  - Rather all the communication is being made through the sidecar proxy.
  - Hence the proxy has all the information regarding the target service's address, how to communicate securely, and what to do if the communica fails.
- Service-mesh is language agnostic: Since the microservice to service mesh proxy communication is always on top to standard protocols such as HTTP1.x/2.x, gRPC etc., one can write microservice from any technology and they will still work with the service mesh.

- A service mesh is usually being coupled with a management and administration utility through which global configuration is being made that is applied to all the sidecar proxies. Moreover, admin can define routes for each services within the mesh using the same utility.

**API Gateway <> Service Mesh ...**

- There is definitely some overlap between API Gateway and Service mesh patterns i.e. request routing, composition, authentication, rate limiting, error handling, monitoring, etc.
- Whereas in actual, the primary focus of a service mesh is service-to-service communication (internal traffic), whereas an API Gateway offers a single entry point for external clients such as a mobile app or a web app used by end-user to access services (external traffic).

**Implementation ...**

- **Linkerd** and **Istio** are two popular open source service mesh implementations.
- They both follow a similar architecture, but different implementation mechanisms.
- Linkerd uses Finagle with Jetty as the sidecar services, whereas Istio uses Envoy.

## Sidecar proxy features ...

A sidecar proxy can provide following features out of the box:

1. **Resiliency for inter-service communications:** Circuit-breaking, retries and timeouts, fault injection, fault handling, fallback and failover.
2. **Service Discovery:** Discovery of service endpoints through a dedicated service registry.
3. **Load balancing:** Client side load balancing to facilitate load distribution on the target service.
4. **Dynamic routing:** Advance routing capabilities, that can base on pre-define policies, partition, and load distribution.
5. **Observability:** Metrics, monitoring, distributed logging, distributed tracing.
6. **Security:** Transport level security (TLS) and key management.
7. **Access Control:** Simple blacklist and whitelist based access control.
8. **Deployment:** Native support for containers. Docker and Kubernetes.
9. **Interservice communication protocols:** HTTP1.x, HTTP2, gRPC

## Implementations ...

| | |
|---|---|
| Lyft's Envoy | https://www.envoyproxy.io/ |
| Netflix's Prana | https://github.com/Netflix/Prana |
| Airbnb's SmartStack | https://github.com/airbnb/smartstack-cookbook |
| Twitter's Finagle | https://github.com/twitter/finagle |

# Resilience | Failover strategies …

**Auto restart**

Microservices either infrastructure or business must be equiped with methods to auto start when their instance/node/host boots-up, restarts, and when the service crashes.

Starting the service on system boots or reboots is straight forward and requires simple configuration. However for restarting the service on crash require custom or third party solutions. For custom solution, one can develop an application that actually spawns the service as a child, and being the parent process monitors the child behavior, and when the child crashes (with unusual exit code), then do the service state cleanup, capture logs and crash dumps if any, and then respawn the service again as child process. For third party solutions, there are many solutions. For instance, one can define auto-start in the docker container conf, in Kubernetes, etc.

With auto-restart configuration, one can live with a single instance of a less frequently used service. Moreover, it is a must have resilience for each and every service within the architecutre.

**Heart beat health checking**

However having autorestart capabilities enables a service to get revive after a crash, but it doesn't rescue it from deadlocks and long starvation scenarios. Hence for that, a more engaging strategy is used called heart beat health checking. It requires a service to emit heart beat signals in the most efficient, effective, and convinent manner to a rescue process. If the rescue process fails to receive these signals for a definite time period, it tear down the current service process, clean up its state, and then restart the process again.

The signaling can be pings over TCP/IP sockets, some file updates on a filesystem, or an OS specific signaling mechanism. But it should emit from the services' processing threads so that the monitoring process can get know whether the main processing thread is working or work.

# Resilience | Technologies, Frameworks, Libraries ...

| Technologies, frameworks ... | | | |
|---|---|---|---|
| **Netflix Ribbon** | Used to implement client side load balancing. | **NginX** | Reverse proxy to manage server side load balancing. |
| **Eureka** | Discovery service from Netflix OSS with Spring Cloud support. | **HAProxy** | Reverse proxy to manage server side load balancing. |
| **Zookeeper** | Discovery service from Apache Foundation with Spring Cloud support. | **Istio** | Provides service mesh implementation. |
| **Consul** | Discovery service from Hashicorp with Spring Cloud support. | **Linkerd** | Provides service mesh implementation. |
| **Hystrix** | Communication resilience wrapper from Netflix provides impl. for circuit breaker, timeout, bulkheads, fallback, etc. | **Envoy** | Sidecar proxy from Lyft. |
| **Resilience4J** | Communication resilience wrapper library that provides impl for various resilient patterns including circuit breaker, bulkheads, etc. | **Prana** | Sidecar proxy from Netflix. |
| **OpenFeign** | A HTTP client from Netflix that provides client side load balancing, and can work with Hystrix. | **Finagle** | Sidecar proxy from Twitter. |
| **Apache HTTPClient** | A HTTP client from Netflix that provides support for basic resilience patterns like timeouts, retrys, etc. | **SmartStack** | Sidecar proxy from Airbnb Engineering. |

# Resilience | Best practices …

- Always embrase **isolation** and **loose coupling** in the design as it always pave the way towards more resilient design.

- Always get know of all possible **failure scenarios** for each of the microservices within the architecture.

- Always use ...
  - solutions and configuration for **auto restart** on services
  - **location transparency** which means accessing others through names, and registrying itself with a name
  - **load balancing** on services that have high load or communication volume
  - HTTP clients that have at least **timeouts** and **retry** patterns implementations

- Always try to use ...
  - **message queues** or **message bus** as the integration strategy, as it makes the communication more resilient and reduces the need for implementing other resilient patterns
  - **event driven design** within the service and service communications
  - **monitoring** and **health check** signaling

- Only use **Hystrix** where it is needed.

- Always assume failures as cases rather than exceptions.

- Never leave a service with a single instance especially if it is being called frequently.

# Logging & Monitoring Strategies ...

A microservice architecture is designed to give full scale automation in **capturing and managing services & infrastructure logs**, and performing **efficient monitoring**.

# Monitoring | Introduction ...

**Microservices are solving many architectural problems** that have plagued developers and operations for years, but using them **increases the system's overall complexity along with the number of failure cases** (as the number of ways a system fails increases with its complexity). Hence one **cannot proactively take care of all the failure cases in such systems**. Then there has to be **mechanisms in place that react to failures** and restore order to the system. These **reaction systems requires constant feedback**. And this **feedback comes from effective & efficient monitoring**.

For a microservice, monitoring means ...

- Understand the overall health of the application
- Determine the aspects of the application that increases the risk of its failure
- Glean insight into the performance of each individual service that makes up an application
- Making sure the services are performing as per expectation based on the load
- Ensure the API transactions are available and performing well
- Isolate problematic transactions and endpoints
- Optimize end-user experience

## Metric ...

Monitoring is a process of reporting, gathering and storing data. This data is usually called **metrics**. A metric is a measure of some process; a number that defines how the process is progressing or failing. It is accompanied with a unit that gives a presice indication of the measure that makes a matric usable in a mathematical decision/process. Moreover, there is also are benchmarks for the metric that defines the worst and optimal measure.

Metrics can be collected from various sources, such as:

- **Application** → These metrics relate specifically to the application. If the application accepts user registrations, a standard metric might be how many were successfully completed in the last hour. Moreover, application's process state also provides several metrics like memory consumption, CPU load, etc.
- **Application's environment** → These are the metrics from the application dependent systems and resources such as database access rate, number of records inserted, number of messages send/dispatched, etc.
- **System's events** → These are OS specific metrics that includes the overall CPU load, I/O load, filesystem availability, network activity, etc. Any unusual hike in these metrics are the signal towards upcoming disaster.
- **Infrastructure** → These metrics report on the nuts and bolts of the infrastructure. These can be the number of instances, load per instance, activity per instance, failure rate, request drop rate, etc.

# Monitoring | Types of monitoring …

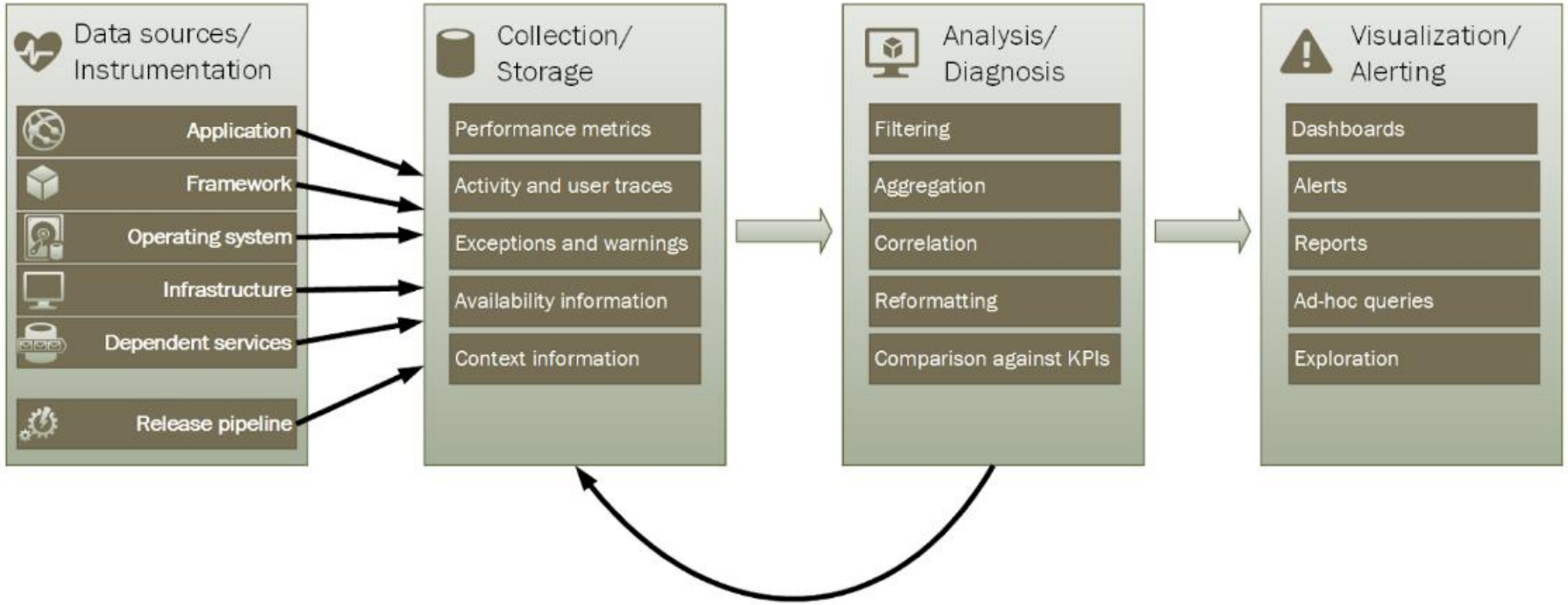
As monitoring is based on metrics, these can be grouped based on their types. These types are as follows:

| | |
|---|---|
| **Health monitoring** | Determine whether the application process state is stable or not. These are the metrics the provide insight into the threads load, memory load, I/O load, etc. A process with measures that show high resource usage means the process is near its failure due to unavailable resources. |
| **Availability monitoring** | Determine whether the application is available to the new requests or not. This requires check the application interfaces for the new requests. If a metric show no or low response measure on an interface, then this means the application is near to become unresponsive. |
| **Performance monitoring** | Determine the application response time to new requests. Performance degradation can be due to various reasons, due to any dependent module/system or due to specific requests. The metrics here provides insight towards the causes of these degradation. |
| **Security monitoring** | These are based on metrics that provide insight into sensitive operations and events. Such as, the number of failed sign-in per source, number of failed queries, etc. This monitoring ensures the security of the system, its services and infrastructure. |
| **Audit monitoring** | Metrics that helps in auditing operations and user activities as part of a monitoring policy or regulation. These may include for instance the number of times a user purchases per unit time, or the amount of expenditure a user has made per unit time, etc. Auditing can provide evidence that links customers to specific requests. Non-repudiation is an important factor in many e-business systems to help maintain trust be between a customer and the organization that's responsible for the application or service. |
| **Usage monitoring** | Its is based on the metrics that provides information on the usage of services. Such as number of time order has made, the number of times purchase was canceled, the number of emails dispatched, the volume used for files per user, etc. This monitoring helps the managers to forecast their system expenses, and scalability needs. |

# Monitoring | Implementing monitoring …

**SO … in monitoring following is done:**

1. Each microservice provides some health APIs through which monitoring system can query the health parameters.
2. The microservices either register themselves to the monitoring server for periodic health checks, or the monitoring server hockup with the registry server to get know the microservices automatically.
3. The monitoring system then call the known microservice health APIs to get know their health parameters, and metrics.
4. The monitoring system will then store the health data to a database and/or send it to a log aggregator server.
5. A analyzer service can then process the health data from the database and can create system events.
6. Configured and important events are then converted into alerts, and these alerts are then dispatched through emails, SMSes, instant-messages, phone-calls, etc.
7. A visualizer application can be used by the human admin to view the system and its contituents performance and failure history.



# Monitoring | Implementing monitoring ...

Microservice monitoring can be implemented using

- **Spring Boot Actuator**
- **Spring Boot Admin**
- **Spring Boot Admin Client**

**Prometheus** can be used as a replacement for the Spring Boot Admin. With that, a **Grafana** log analyzer can be used. Similarly, **Metricbeat** can also be used to collect various system, and service metrics and dump them on an external database.

Besides that, Eureka based discovery server and API Gateway can be configured to provide Spring admin dashboards. Make sure API Gateway has Hystrix dashboard annotation defined.

https://prometheus.io/

https://grafana.com/

**Prometheus + Grafana ...**

- Prometheus is more sophisticated and feature rich alternative to Admin server. It provides a lot of configurable integrations and storage options.
- Grafana is a log analysis system, through which one can unify monitoring, define alerts, dispatch notifications, and visualize raw, and processed data.
- See for more details ...

https://g00glen00b.be/monitoring-spring-prometheus-grafana/

https://spring.io/guides/gs/actuator-service/

**Actuator API ...**

- Actuators capture the microservice (Spring Boot app) performance metrics and then made them available on an REST resource.
- To use, one will have to add its dependency in the project file.
- Enable the actuators in the application properties/yml file.

**Admin Server ...**

- This is a small monitoring server that can capture all the metrics exposed by Actuators in the microservices, and have them display in a single dashboard.
- Either create a separate microservice, or piggyback it with the discovery server, and then define the admin server dependency in the project file.
- If a seperate server is being created, then each and every microservice will have to define the admin server access configuration. Whereas if it is bundled with the discovery server, it can get know about all the microservices from there; and hence no need for the microservices to configure admin server.
- It currently supports Eureka, Zookeeper, and Consul (as discovery servers) to work along with.
- Moreover, in the main application class, define the admin server annotation.
- And finally define the admin REST resource in the application properties/yml file.

**Admin Client ...**

- This is admin server accompany module for the microservice that want monitoring. It makes the microservice to post its metrics periodically to the admin server.
- To use, one has to define its dependency in the project file.
- Define the admin server access configuration in the application properties/yml file.

# Logging | Logging in microservice architecture ...

In microservice architecture, **centralized logging** is implemented with **service tracing**.

## Centralized logging ...

In microservice architecture, the services are usually deployed in expandable containers. These containers are created and tear down based on the scalability needs. The services logs are created within the container environment. In order to view these logs, one will then has to log into the container, and then view the logs. In order to view all logs, one will then has to open up log files for every instance of every service. And then determining a single operation/workflow from these log entries make the audit an impossible job.

The solution is to implement centralized logging. In this design the logs from the services are aggregated on a single server which then store these logs in a specialized (usually NoSQL) database for later query.

Hence the log aggregator becomes another infrastructure service that also do other logs processing, like transforming logs to structure data, filtering logs, and grouping them.

The logs aggregator stores the data in a format that makes it easy for the log analyzer application to pull out logs and create knowledge on top of it.

## Distributed Tracing ...

It is a process of grouping logs from several services based on some correlation ID.

Even with centralized logging, one cannot make sense of all the logs in a single place. And it is only due to the fact that one cannot distinguish a single operation/workflow in the logs that happens across multiple services. Hence logs for all the operations are mixed together, and there is no way to take out logs entries for a single operation.

Distributed tracing provides a way of tagging each logs entry with an ID called correlation ID. Log entries that belongs to the same operation are assigned with same ID. Hence getting log entries for an operation becomes simple as one has to query records for that correlation ID.

Distributed tracing is some time also called service tracing.

# Logging | Log types & processes …

- In a microservice architecture, there are different types of logs:
  1. Application logs
  2. Service trace logs
  3. Infrastructure logs
  4. Application performance logs
- There can be more types of logs besides the mentioned above.

- These logs are generated by the microservices, and are usually captured in a difference way.
- The captured logs are then aggregated, grouped and stored on a separate store for easy viewing, and for generating reports on top of it.

- Hence, there are five types of processes involved with logs generation:
  1. Logs generator
  2. Logs capturing
  3. Logs aggregation
  4. Logs storage
  5. Logs analyzers

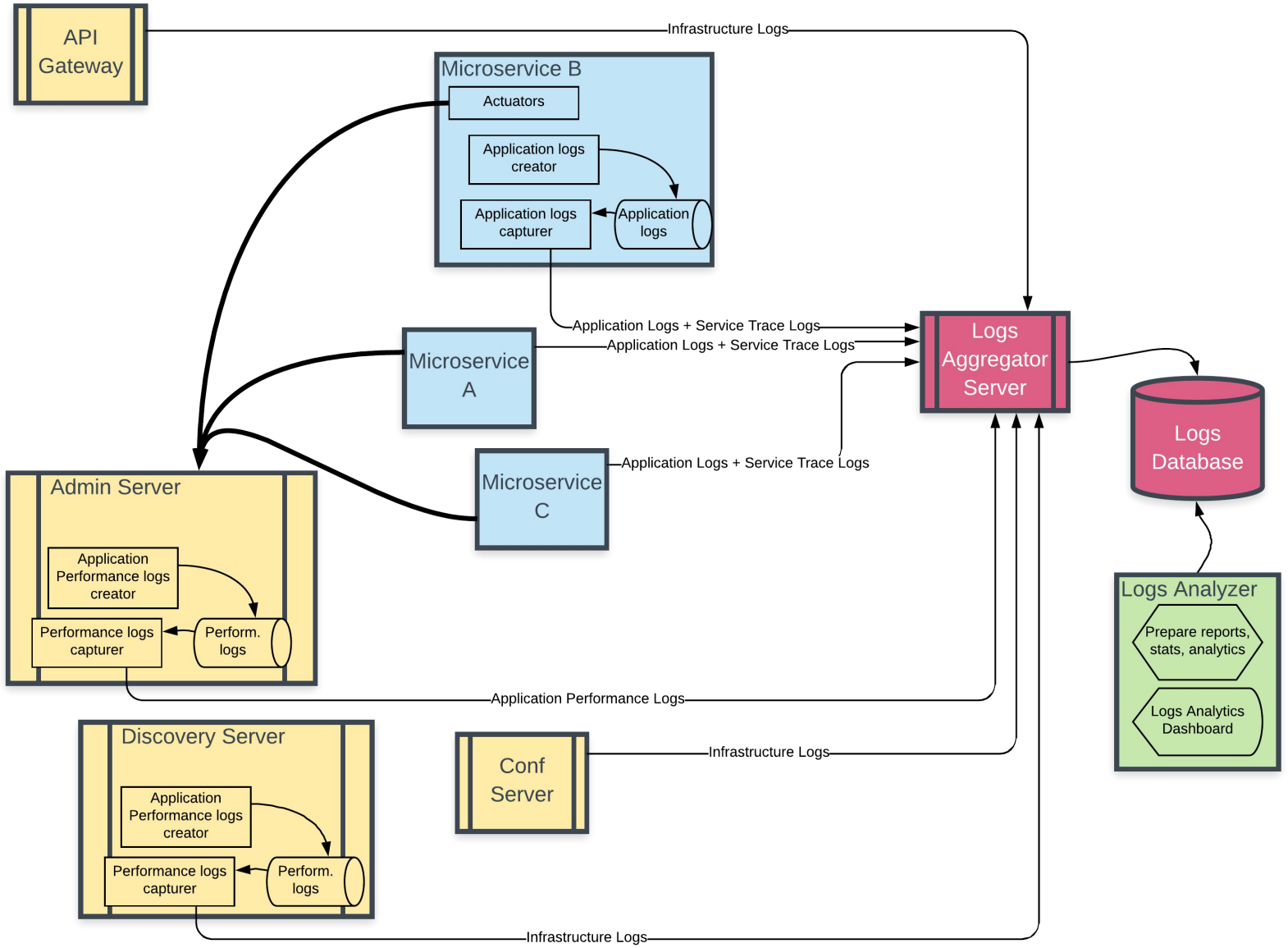
**Application logs** are the operation logs created by the business microservices that captures the business operation being executed in the service. These logs are usually being created in the service instance filesystem.

**Service trace logs** are the group of logs that captures the incoming requests, and the outgoing responses from all the services that are engage in a request workflow stream. These logs besides being saved to the application logs, can also be send to a centralized aggregator service.

**Infrastructure logs** are the application logs created from the microservices, servers, applications that are part of the infrastructure.

**Application performance logs** captures the performance metrics for business microservices like the CPU load, mem usage, network activties, etc. These logs can be generated by the application itself, or by a central admin server.

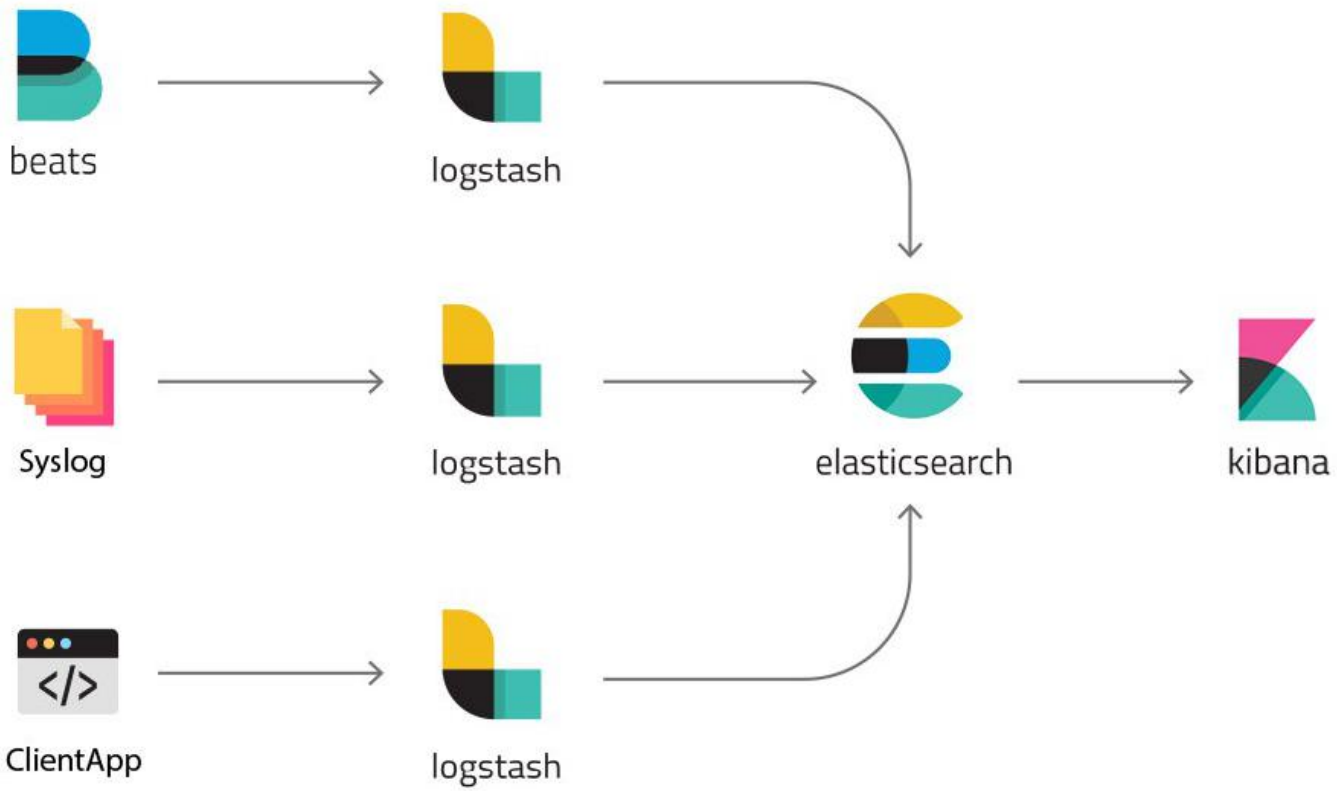
# Logging | Logging architecture ...




Application level logs can be captured and store in a database in different ways. Some are:

1. Use logs capturer software to get the logs, and have them transmitted to an external aggregator. For instance, Zapkin, Logstash, ...
2. Use a specialized SLF4J logback appender to directly transmit the logs to the logs aggregator server.
3. Use a specialized SLF4J logback appender to directly write logs to the logs database.
4. Use a logs service client SDK/API/library to write logs directly to the online service.

# Logging | Logging with ELK ...




## ELK Stack ...

ElasticSearch, LogStash, and Kibana make up the ELK stack.

- **Elasticsearch** is a search engine, that provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents.
- **Logstash** is a plugin-based data collection and processing engine. It comes with a wide range of plugins that makes it possible to easily configure it to collect, process and forward data in many different architectures. Processing is organized into one or more pipelines. In each pipeline, one or more input plugins receive or collect data that is then placed on an internal queue. This is by default small and held in memory, but can be configured to be larger and persisted on disk in order to improve reliability and resiliency.
- **Kibana** is an open source data visualization plugin for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bar, line and scatter plots, or pie charts and maps on top of large volumes of data.

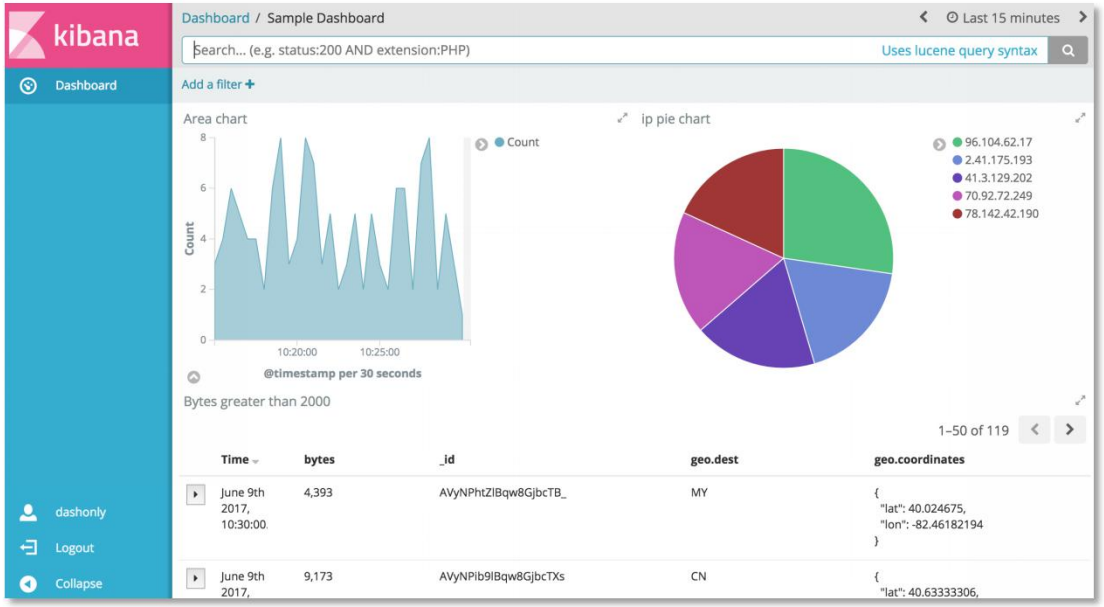
## Two ways to capture logs through ELK ...

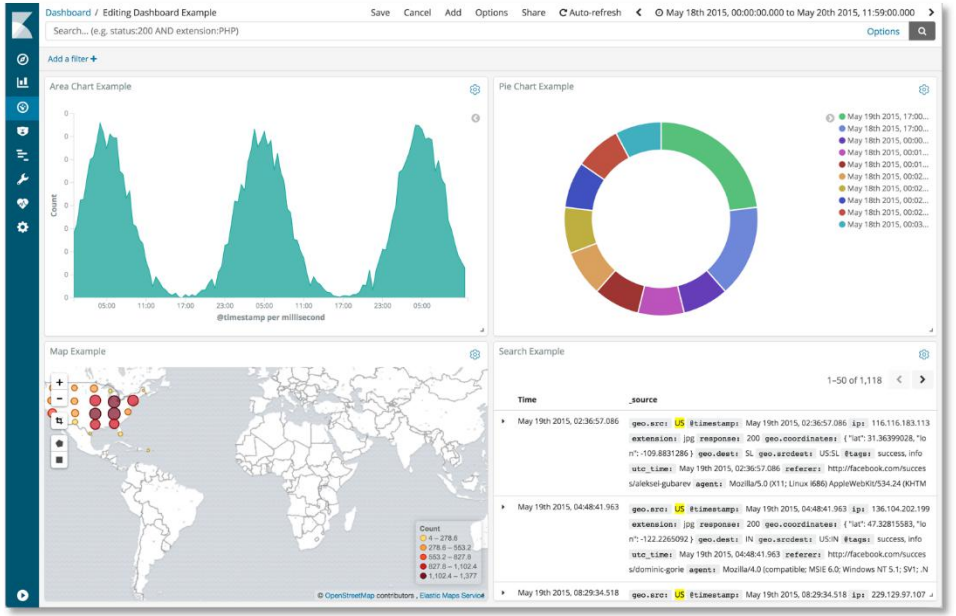
| | |
|---|---|
| 1) Configure **FileBeats** at the microservice environment, and configure it to read log files, and then transmit to logstash over TCP/IP socket. | 2) Configure microservice logs configuration, and use logstash appender to directly transmit logs entries to the logstash over TCP/IP socket. |

# Logging | Logging technologies ...

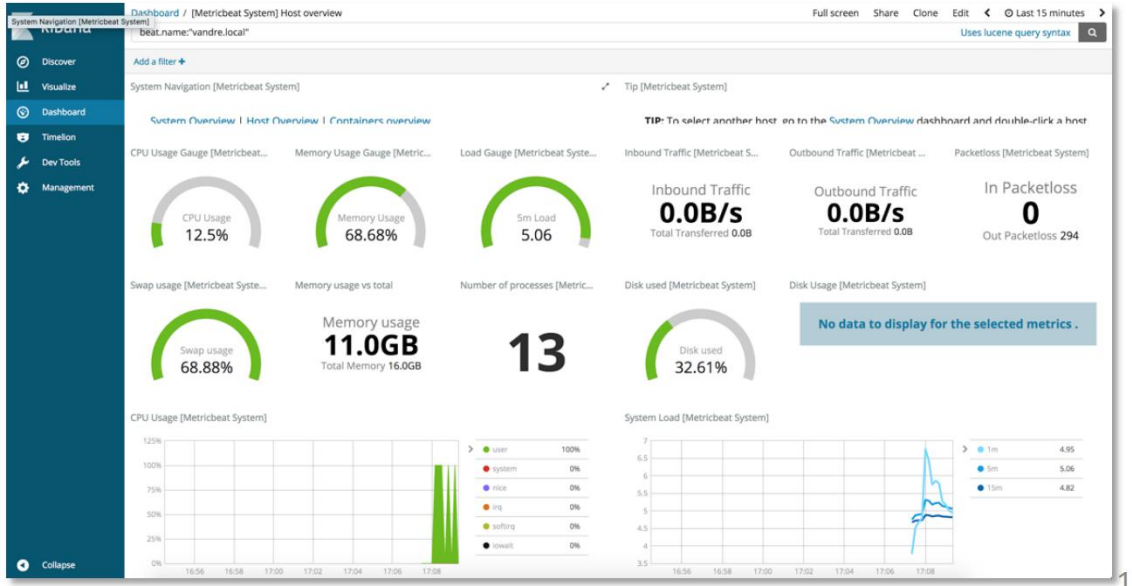
| Type | Tech | Purpose |
|---|---|---|
| Logs generator | SLF4J, Logback, Log4J, Spring Boot Sleuth, ... | To create logs for each and every business operation, activity happens in a microservice for a incoming request. |
| Logs capturer | Logstash's Filebeat, Fluentd, ... | To read application logs, and filter then, process them, transform them, and finally send them to an aggregator service. |
| Logs aggregator | Zapkin, Logstash, Dapper, ... | An external server that receives logs from various sources in the microservice architecture, and it then tries to group them and then store them on a preconfigured storage. |
| Logs store | MySQL, Cassandra, Elasticearch, ... | These are the database tech that actually stores the logs with a structure for later analysis. Its choice is made on the usage pattern, and its capabilities for that. |
| Logs analyzers | Kibana, Splunk, ... | These are the external applications that are run on the logs data in the logs store, and hence based on the configuration generates reports, statistics, and analysis. |
| Logs services | Timber.io, LogDNA, logDog, PaperTrail, ... | These are the online services accessible through Internet. Microservices can send there logs directly to them. Hence they are the one stop shop that does the capturing, aggregation, storage, and analysis. Moreover, the logs can be access from anywhere without being access to the infrastructure. |

# Logging | Kibana dashboard …



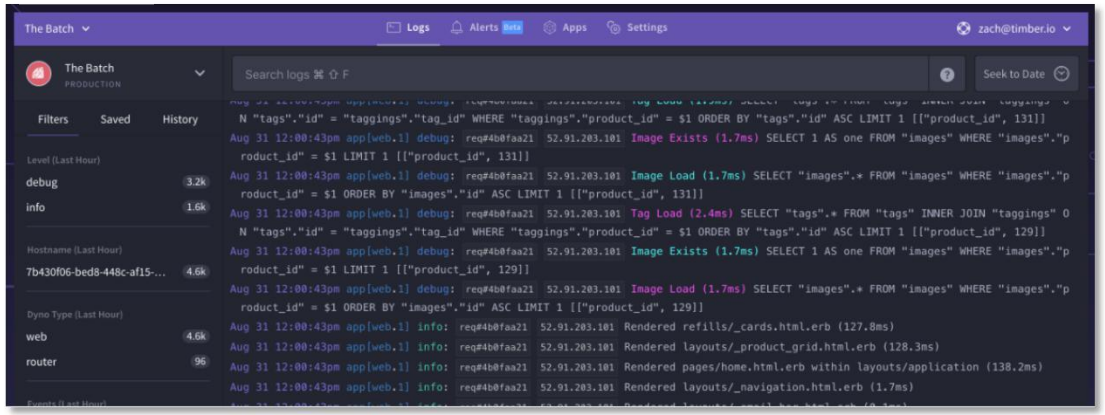



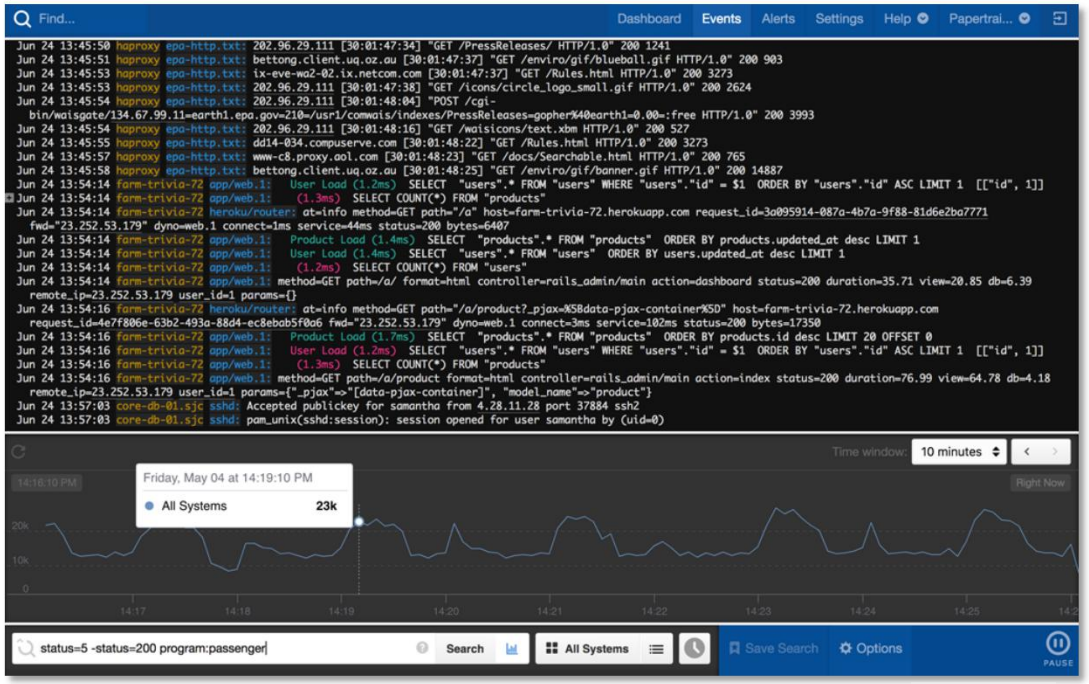




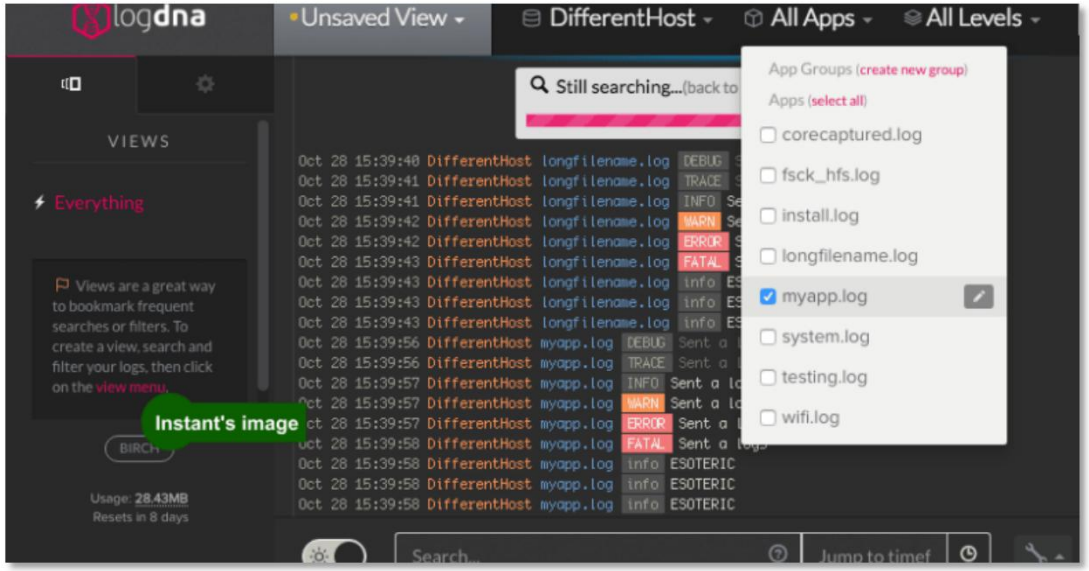
# Logging | Online services dashboards …

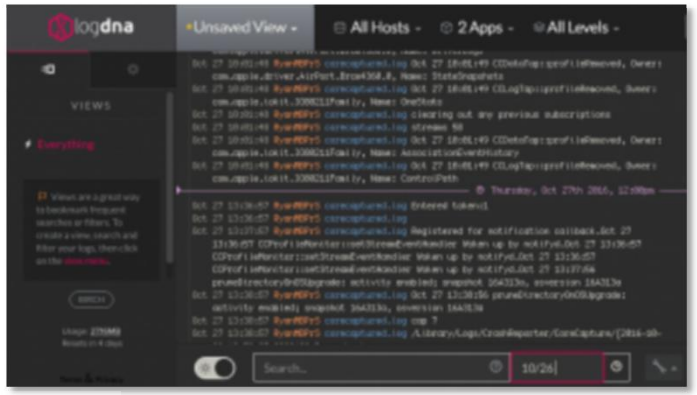
Timber.io




PaperTrail




LogDNA

# Logging | Best practices ...

## Monitoring ...

- Always implement **health check APIs** in the microservice so that proper application metrics can be collected by the monitoring system.
- Nobody has time to continously watch monitoring logs. Create **rules**, **events**, and **alerts** to get notify on the unuasual, unwanted scenarios/situations/events.
- Monitoring is not only for the system and its service. Extend monitoring to the environment, its resources, and infrastructure.
- Identify failure scenarios, and then **metrics** that can predict the risk or probability of their occurances.
- Half of the security will be based on monitoring; so make sure to setup metrics that helps in **security scrutiny**, and **audits**.

## Logging ...

- Always enforce practice to put logs in the microservices codes as part of the development process, such that
  - it records logs using proper logs levels,
  - it should capture call traces, debug info, successful and failure cases, unuaual situations, scenarios that causes exceptions, recoverable failures, and non-recoverable failures.
- Always use **SLF4J** as the logging framework.
- Always enable **distributed tracing**.
- Either in the start or later, but always plan for **centralized logging**.

# Deployments models ...

A microservice architecture effectiveness depends on its deployment strategy. The service level resilience, and fault tolerance is offen get from how the service is being deployed.

# Deployment | Deployment Patterns ...

Microservices can be deployed using any of the deployment strategies ...

| | |
|---|---|
| Multiple service instances per host | • Deploying all microservice instances on a single physical machine (host), or on multiple hosts.<br>• Easy, and usually used by developers at development stage. |
| Service instance per host | • Deploying every microservice instance on a separate physical machine.<br>• Expensive, and legacy way of doing production on-prem deployment. |
| Service instance per VM | • Setting up a virtualization server, and deploying microservice instances on separate virtual machines (VMs).<br>• An old way of having production deployments. Still used today in several monolith software deployments.<br>• Some examples are **VMware Virtualization Server**, **Citrix Server**, **Oracle VirtualBox**, etc. |
| Service instance per container | • Setting up application container platform on one or more physical or virtual machines, and deploying each microservice instance on a separate app container.<br>• A new way of having production deployments.<br>• Some exmples are **Docker**, **Rocket**, **LXD**, **OpenVZ**, **LinuxVServer**, **Windows Containers**, etc. |
| Serverless deployment | • Use a deployment infrastructure that hides any concept of servers (i.e. reserved or preallocated resources)- physical or virtual hosts, or containers. The infrastructure takes the service's code and runs it.<br>• Hence the business concern is developed as a callable module, and it is then deployed on the infrastructure, and the module entry point is then bind with an API or an event.<br>• For exmple **AWS Lamda**, **GCP's Cloud Functions**, etc. |
| Service deployment platform | • Setting up a deployment platform that take up the microservices in a definite way, and then provide an automated way of deployment, management and orchestration.<br>• The deployment platform provides implementations for all the distributed system concerns, and micorservice patterns, thus leaving the microservices to only conern with the business logic.<br>• It is latest form of deployment strategy that truely enables scalable deployment that can support large number of users, high volume data, and transactions.<br>• For example **Kubernetes**, **Heroku**, **Docker Swarm**, **Apache Mesos**, etc. |

Now a days, services are being deployed as containers using deployment platform that provides a homogenous management experience in all the cloud platforms ...

# Deployment | Application Containers

A container is a standard unit of software → packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

- Hence a container is just an environment for the target application that contains the application, its runtime, its dependent libraries, configurations, and other resources.
- A container usually shares OS and its system libraries with the host environment.
  - In case of having containers on a different OS, then emulation is being used. For instance setting up Linux based containers on Windows OS.
- An application in a container with the matching host OS gives almost the same performance as it is deployed directly on the host environment.
- An containerized application can be serialized in an image, and then the image can be replicated and deployed on other hosts as per need.
  - Hence a service containerized can be saved as an image, and then can be used later for repeated number of deployments.

- Containers provides isolation to the application such that
  - an application cannot see others in the other containers
  - an application cannot impact / affect others applicaitons
  - an application cannot access resources in other application containers
  - an application can have dedicated runtime and dependencies for specific version

## History of containers ...

FreeBSD Jails (2000) ... FreeBSD Jails allows administrators to partition a FreeBSD computer system into several independent, smaller systems – called “jails” – with the ability to assign an IP address for each system and configuration.

Linux VServer (2001) ... Linux VServer is a jail mechanism that can partition resources (file systems, network addresses, memory) on a computer system. Introduced in 2001, this operating system virtualization that is implemented by patching the Linux kernel.

Solaris Containers (2004) ... Solaris Containers combines system resource controls and boundary separation provided by zones, which were able to leverage features like snapshots and cloning from ZFS.

OpenVZ (2005) ... This is an operating system-level virtualization technology for LinuopenVZ history of containersx which uses a patched Linux kernel for virtualization, isolation, resource management and checkpointing.

Process Containers (2006) ... Process Containers (launched by Google) was designed for limiting, accounting and isolating resource usage (CPU, memory, disk I/O, network) of a collection of processes. It was renamed “Control Groups (cgroups)” a year later and eventually merged to Linux kernel 2.6.24.

LXC (2008) ... LXC (LinuX Containers) was the first, most complete implementation of Linux container manager. It uses cgroups and Linux namespaces, and it works on a single Linux kernel without requiring any patches.

Warden (2011) ... Warden started by CloudFoundry, uses LXC in the early stages and later replacing it with its own containers implementation. Warden can isolate environments on any operating system, running as a daemon and providing an API for container management. It developed a client-server model to manage a collection of containers across multiple hosts, and Warden includes a service to manage cgroups, namespaces and the process life cycle.

LMCTFY (2013) ... Let Me Contain That For You (LMCTFY) kicked off in 2013 as an open-source version of Google's container stack, providing Linux application containers. Applications can be made “container aware,” creating and managing their own subcontainers.

Docker (2013) ... *Detail discussion in later slides*.

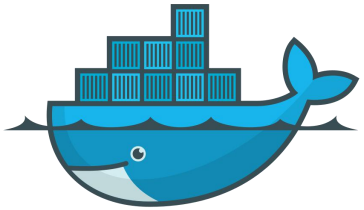
# Deployment | (cont.) Application Containers

Benefits ...

- **Agile application creation and deployment:** Increased ease and efficiency of container image creation compared to VM image use.
- **Continuous development, integration, and deployment:** Provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
- **Dev and Ops separation of concerns:** Create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- **Observability:** Not only surfaces OS-level information and metrics, but also application health and other signals.
- **Environmental consistency across development, testing, and production:** Runs the same on a laptop as it does in the cloud.
- **Cloud and OS distribution portability:** Runs on Ubuntu, RHEL, CoreOS, on-prem, Google Kubernetes Engine, and anywhere else.
- **Application-centric management:** Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- **Loosely coupled, distributed, elastic, liberated micro-services:** Applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- **Resource isolation:** Predictable application performance.
- **Resource utilization:** High efficiency and density.

# Deployment | Docker

https://www.docker.com



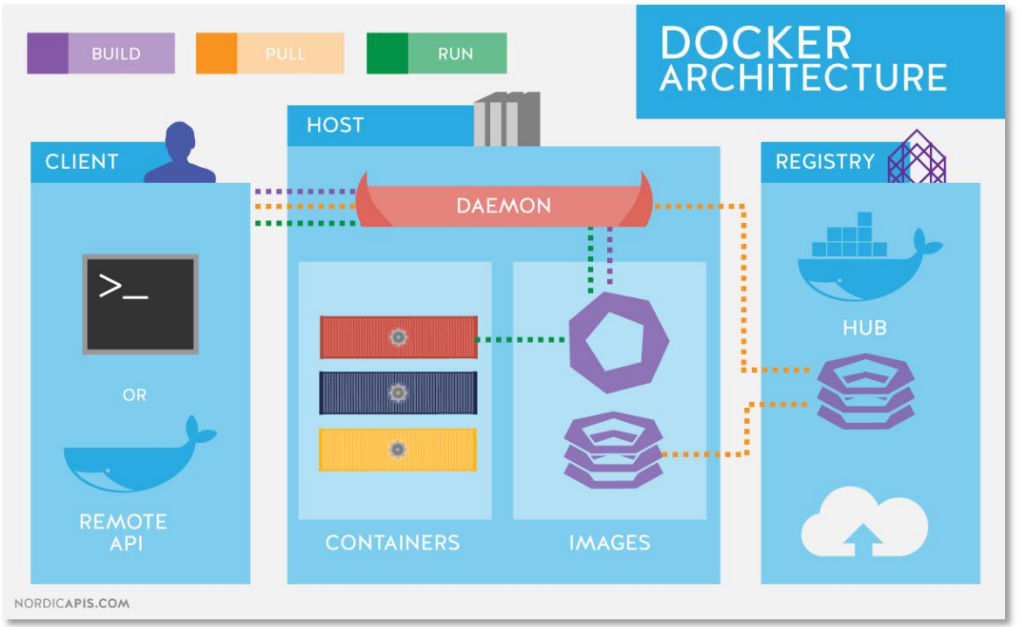
Docker is an application container software that not only provides the capability of a containerization, but also management. It was first authored by Solomon Hykes, later released as open source in March 2013. On March 13, 2014, with the release of version 0.9, Docker dropped LXC as the default execution environment and replaced it with its own libcontainer library written in the Go programming language.

Docker is developed primarily for Linux, where it uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file system such as OverlayFS and others to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines (VMs). The Linux kernel's support for namespaces mostly isolates an application's view of the operating environment, including process trees, network, user IDs and mounted file systems, while the kernel's cgroups provide resource limiting for memory and CPU. Since version 0.9, Docker includes the libcontainer library as its own way to directly use virtualization facilities provided by the Linux kernel, in addition to using abstracted virtualization interfaces via libvirt, LXC and systemd-nspawn. A Docker container, unlike a virtual machine, does not require or include a separate operating system.

## Docker Architecture ...

To allow for an application to be self-contained the Docker approach moves up the abstraction of resources from the hardware level to the Operating System level. To further understand Docker, let us look at its architecture. It uses a client-server model and comprises of the following components:

- **Docker daemon:** The daemon is responsible for all container related actions and receives commands via the CLI or the REST API.
- **Docker Client:** A Docker client is how users interact with Docker. The Docker client can reside on the same host as the daemon or a remote host.
- **Docker Objects:** Objects are used to assemble an application. Apart from networks, volumes, services, and other objects the two main requisite objects are:
- **Images:** The read-only template used to build containers. Images are used to store and ship applications.
- **Containers:** Containers are encapsulated environments in which applications are run. A container is defined by the image and configuration options. At a lower level, you have containerd, which is a core container runtime that initiates, and supervises container performance.
- **Docker Registries:** Registries are locations from where we store and download (or “pull”) images.



# Deployment | (cont.) Docker

## Docker File ...

**Dockerfile defines what goes on in the environment inside a container.**

Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of the system, so one need to map ports to the outside world, and be specific about what files are required to “copy in” to that environment. However, after doing that, one can expect that the build of the app defined in this Dockerfile behaves exactly the same wherever it runs.

A docker file is named as *Dockerfile*, and the docker CLI can be use to build it into an image. The image is then can be use to run the packaged software, or publish it on docker hub repository.

## Docker software services ??

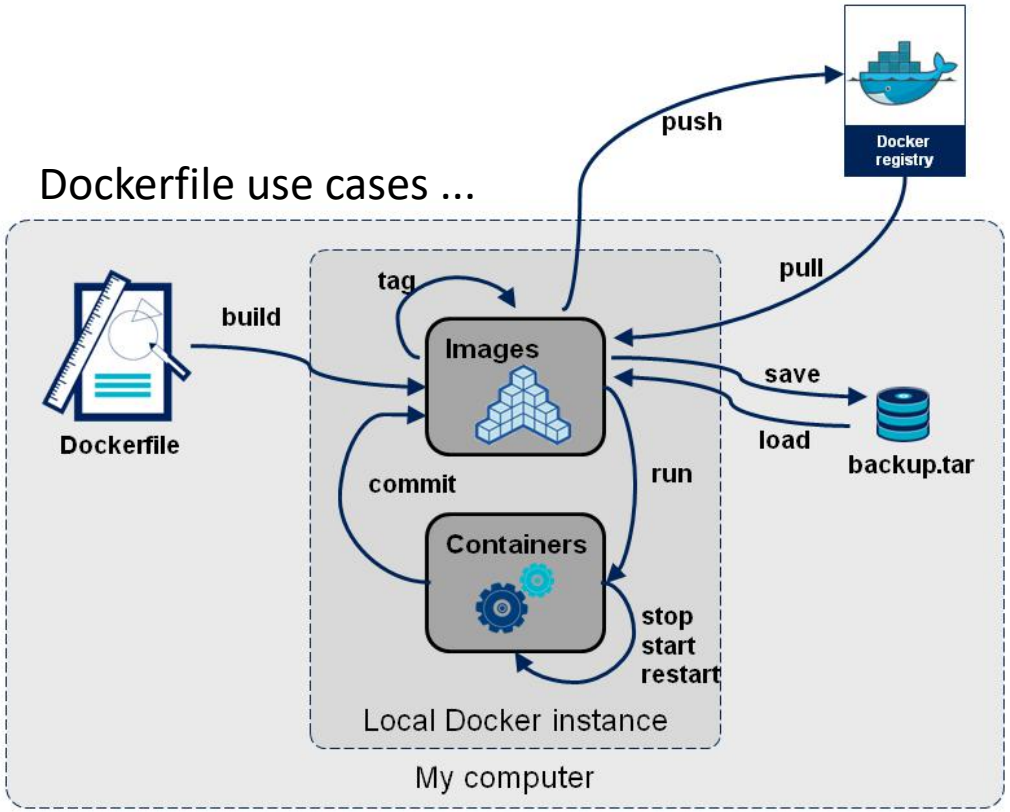
Docker software provides the following base services:

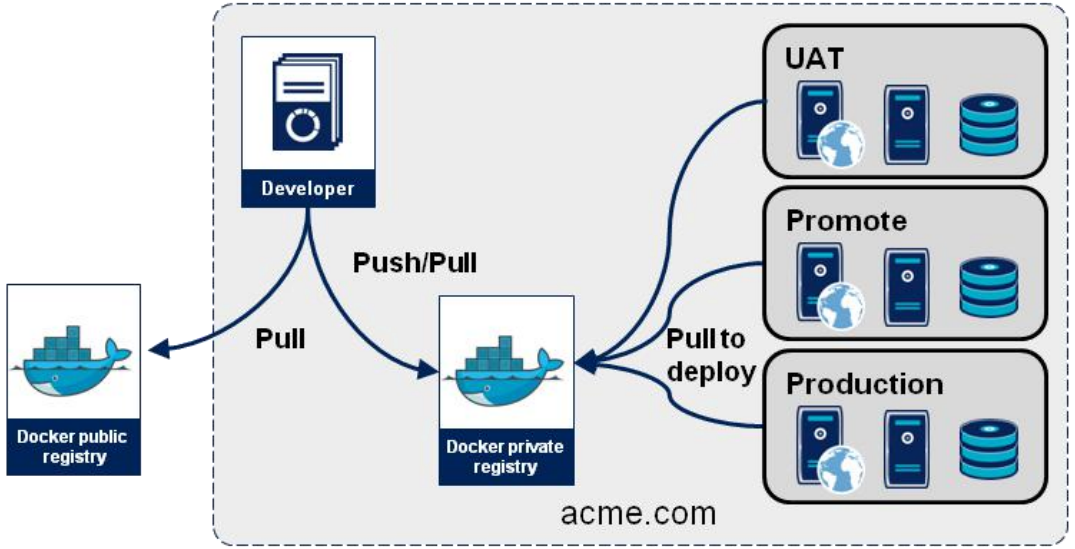
- **App container** → Containerization of applications with support for customizable resources.
- **Image builder** → Ability to build container images that can be used later on same of different host.
- **Container management & deployment** → Provides full control and administration on the deployment and management process.

It also provide other secondary yet important services like:

- **Docker-compose** → Enables easy deployment and management of multiple containers on a single host.
- **Docker Hub** → Provides image respository and distribution services. Hence can be use to publish images, and deploy them anywhere in the world.
- **Docker Swarm** → Allows management of containers on multiple hosts.

Dockerfile use cases ...



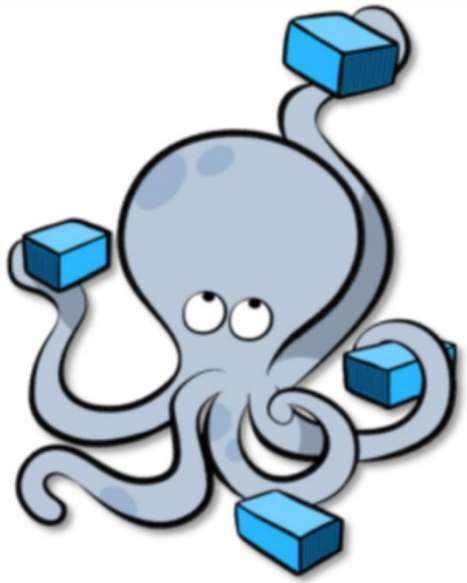

Dockerfile; create once, use many times ...

# Deployment | How Docker can be used in microservice architecture?

## Datacenter as one machine

**Docker-compose is one of the important tool build on docker system that allows automated deployment and management of multiple containers.**

Docker Compose is a tool for defining and running multi-container Docker applications. It uses YAML files to configure the application's services and performs the creation and start-up process of all the containers with a single command. The docker-compose CLI utility allows users to run commands on multiple containers at once, for example, building images, scaling containers, running containers that were stopped, and more. Commands related to image manipulation, or user-interactive options, are not relevant in Docker Compose because they address one container. The docker-compose.yml file is used to define an application's services and includes various configuration options. For example, the build option defines configuration options such as the Dockerfile path, the command option allows one to override default Docker commands, and more.



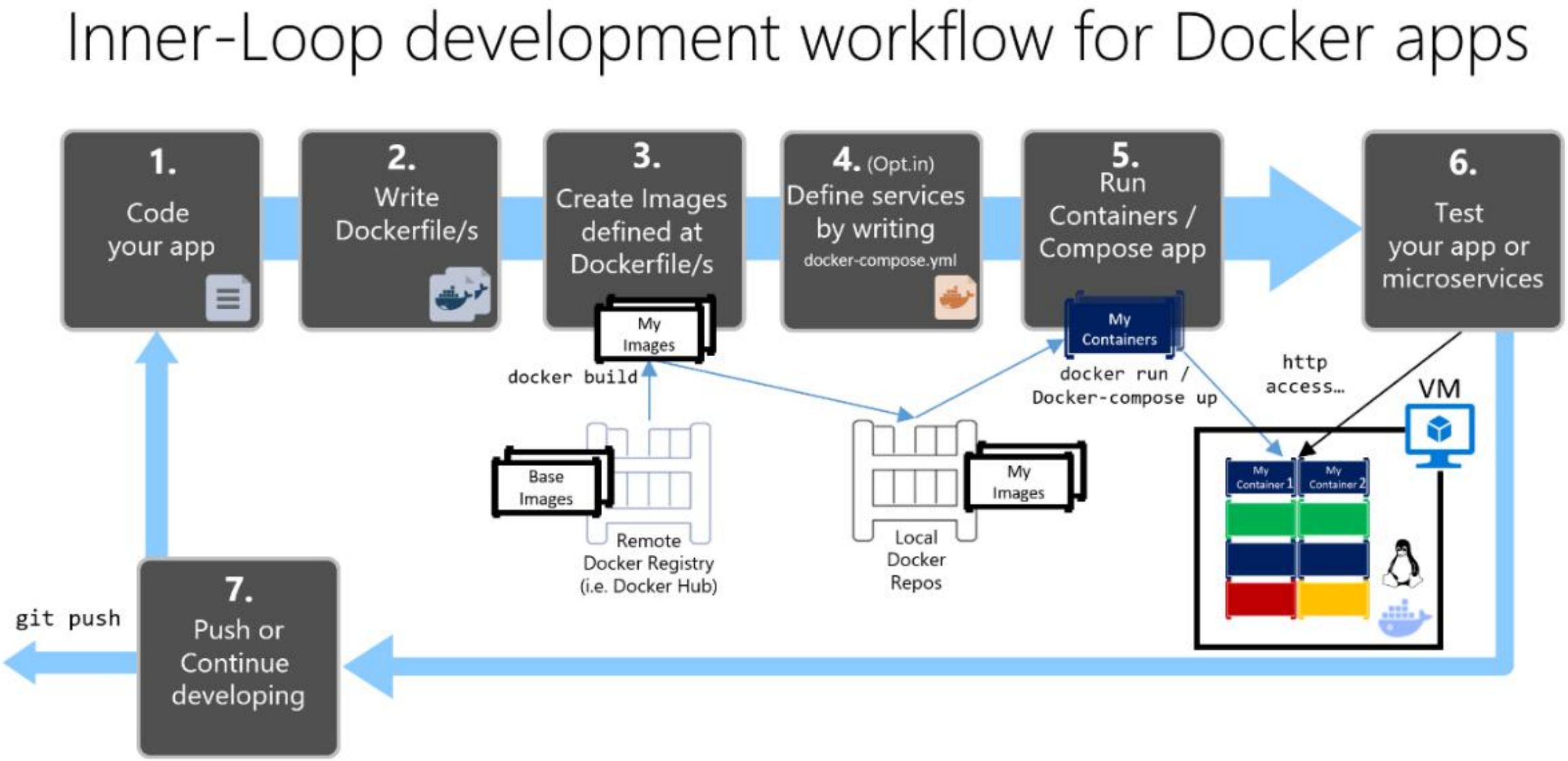
One a single machine, using docker compose, one can define containers for each microservice. The compose yml file will contains all the configuration for launching each container in a specific order, and whether to auto-restart it if it fails. Moreover, multiple containers can be created for a single service to distribute load (load balancing) and for fault tolerance.

## Creating Docker builds

**Docker plugins for maven and gradle project managers are available; that can be use to create docker images directly as the code build artifacts.**

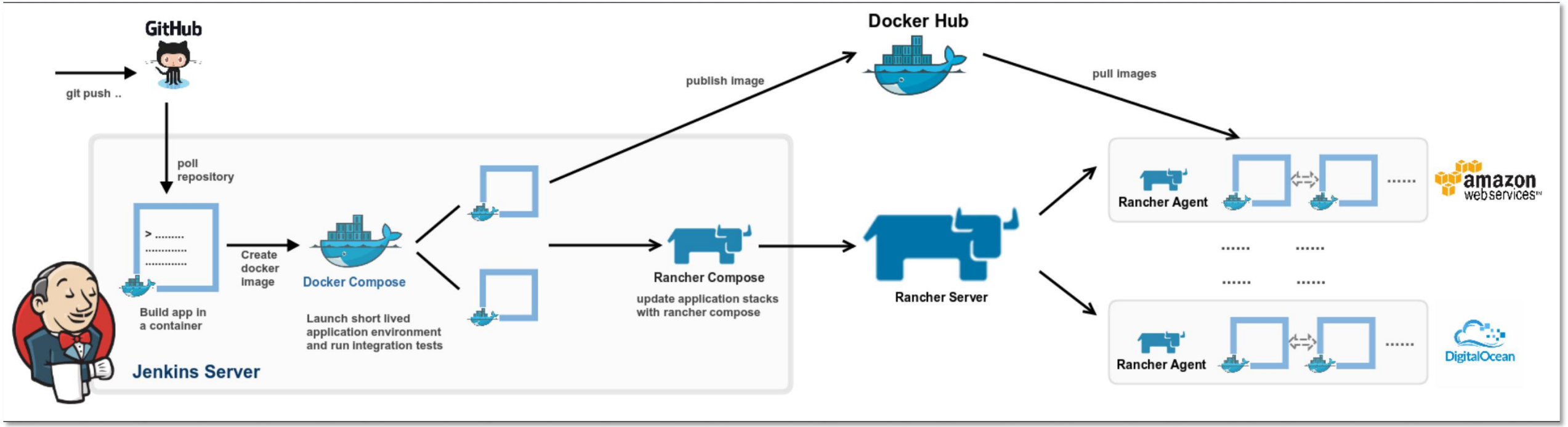
- A SpringBoot microservice application can be build directly as a docker container image.
- All is required to add docker plugin in the POM or gradle project file, and define Dockerfile within the code directly; usually src/main/docker.
- The Dockerfile defines the JDK runtime, filesystem volume, the Java application JAR files, and the calling arguments.
- With that, the build process (maven or gradle) built the target builds, and then generate the docker image file.
- Hence docker images can be generated for each and every microservice.
- These images can then be publish to the Docker registry or docker hub for distribution.

# Deployment | Engineering life cycle with Docker ...




https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/docker-application-development-process/media/image1.png

# Deployment | Microservice deployment automation ...



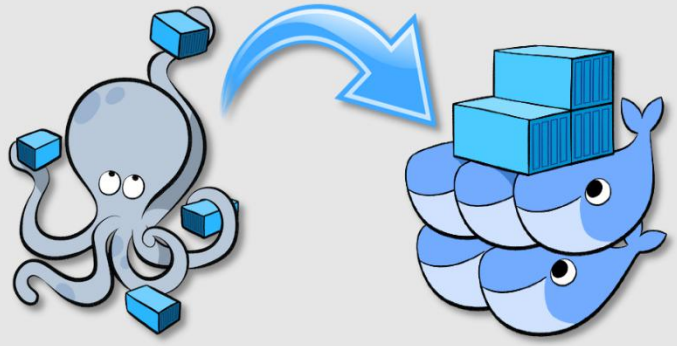

| Lifecycle ... | | |
|---|---|---|
| 1. Development (Engineer) | 2. Push to repository (Engineer) | 3. Pull changes (CI; Jenkins) |
| 4. Create build (CI; Jenkins) | 5. Test artifact (CI; Jenkins) | 6. Create Docker image (CI; Jenkins) |
| 7. Deploy image for test (CI; Jenkins) | 8. Push to Docker registry (CI; Jenkins) | 9. Get new image (CD; Rancher) |
| 10. Deploy new image (CD; Rancher) | 11. Test new image (CD; Rancher) | 12. Remove old image (CD; Rancher) |

# Deployment | Container orchestration …

## Containers on multiple hosts

Docker-swarm is the tool that enables container management on more than one machine. It is one of the **container orchestration solution**.

Docker Swarm provides native clustering functionality for Docker containers, which turns a group of Docker engines into a single virtual Docker engine. In Docker 1.12 and higher, Swarm mode is integrated with Docker Engine. The swarm CLI utility allows users to run Swarm containers, create discovery tokens, list nodes in the cluster, and more. The docker node CLI utility allows users to run various commands to manage nodes in a swarm, for example, listing the nodes in a swarm, updating nodes, and removing nodes from the swarm.



Hence using swarm, one can extend the microservice deployment to multiple hosts, where each host will be have docker engine setup with swarm configuration enabled.

## Orchestration Solutions …

| | |
|---|---|
| Kubernetes | Originally developed by Google as an offshoot of its Borg project, Kubernetes has established itself as the de facto standard for container orchestration. It's the flagship project of the Cloud Native Computing Foundation. |
| Docker Swarm | Even though Docker has fully embraced Kubernetes as the container orchestration engine of choice, the company still offers Swarm, its own fully integrated container orchestration tool. Slightly less extensible and complex than Kubernetes, it's a good choice for Docker enthusiasts who want an easier and faster path to container deployments. In fact, Docker bundles both Swarm and Kubernetes in its enterprise edition in hopes of making them complementary tools. |
| Apache Mesos (and Marathon) | Apache Mesos, slightly older than Kubernetes, is an open source software project originally developed at the University of California at Berkeley, but now widely adopted in organizations like Twitter, Uber, and Paypal. Mesos' lightweight interface lets it scale easily up to 10,000 nodes (or more) and allows frameworks that run on top of it to evolve independently. Its APIs support popular languages like Java, C++, and Python, and it also supports out-of-the-box high availability. Unlike Swarm or Kubernetes, however, Mesos only provides management of the cluster, so a number of frameworks have been built on top of Mesos, including Marathon, a "production-grade" container orchestration platform. |

**Container orchestration** is all about managing the lifecycles of containers, especially in large, dynamic environments with multiple hosts/nodes. Software teams use container orchestration to control and automate many tasks:

- Provisioning and deployment of containers
- Defining communication paths
- Defining resilience schemes
- Designing security parameter
- Redundancy and availability of containers
- Scaling up or removing containers to spread application load evenly across host infrastructure
- Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- Allocation of resources between containers
- External exposure of services running in a container with the outside world
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts
- Configuration of an application in relation to the containers running it

kubernetes MESOS MARATHON

# Microservice Architecture on Steroids …

C2C and B2C systems requires highest level of scalability and resiliance, and with microservice architecture, it is only possible with an dynamic orchestration platform that can support cloud, and has capability of auto scaling and auto management. Enter the world of **Kubernetes; An steroid for the microservice architecture**.

# Kubernetes | What it is anyway?

The name **Kubernetes** originates from Greek, meaning helmsman or pilot, and is the root of governor and cybernetic. **K8s** is an abbreviation derived by replacing the 8 letters "ubernete" with "8".

https://kubernetes.io/

## Key concepts ...

Here's what makes up a Kubernetes cluster:

- **Master**: by default, a single master handles API calls, assigns workloads and maintains configuration state.
- **Minions**: the servers that run workloads and anything else that's not on the master.
- **Pods**: units of compute power, made-up of one or a handful of containers deployed on the same host, that together perform a task, have a single IP address and flat networking within the pods.
- **Services**: front end and load balancer for pods, providing a floating IP for access to the pods that power the service, meaning that changes can happen at the pod-level while maintaining a stable interface.
- **Replication controllers**: responsible for maintaining X replicas of the required pods.
- **Labels**: key-value tags (e.g. "Name") that you and the system use to identify pods, replication controllers and services.

Kubernetes starts big. Whereas Docker Swarm starts at the container and builds out, Kubernetes starts at the cluster and uses containers almost as an implementation detail.

**Kubernetes** is a container orchestration platform that spun out of Google to automates the process of deploying and managing multi-container applications at scale. While Kubernetes works mainly with Docker, it can also work with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes. And because Kubernetes is open source, with relatively few restrictions on how it can be used, it can be used freely by anyone who wants to run containers, most anywhere they want to run them.

It provides high-level abstractions for managing groups of containers that allow Kubernetes users to focus on how they want applications to run, rather than worrying about specific implementation details. The behaviors they need are decoupled from the components that provide them. It is designed to automate and simplify a number of container management tasks.

Some of its key functions:

- Deploys multi-container applications
- Scales containerized apps automatically
- Rolls out new versions of apps without downtime
- Provides networking, service discovery, and storage
- Provides monitoring, logging, and debugging
- Allos for non-destructive customizations
- Operates in any environment

# Kubernetes | Kubectl & control plane ...

## KubeCtl ...

In Kubernetes, the user uses Kubernetes API objects to describe his cluster's desired state → what applications or other workloads he wants to run, what container images they use, the number of replicas, what network and disk resources you want to make available, and more. He set his desired state by creating objects using the Kubernetes API, typically via the command-line interface, **kubectl**. He can also use the Kubernetes API directly to interact with the cluster and set or modify your desired state. Once he has set his desired state, the Kubernetes Control Plane works to make the cluster's current state match the desired state. To do so, Kubernetes performs a variety of tasks automatically–such as starting or restarting containers, scaling the number of replicas of a given application, and more. The Kubernetes Control Plane consists of a collection of processes running on the cluster.

## Control Plane ...

The various parts of the Kubernetes Control Plane, such as the Kubernetes Master and kubelet processes, govern how Kubernetes communicates with the cluster. The Control Plane maintains a record of all of the Kubernetes Objects in the system, and runs continuous control loops to manage those objects' state. At any given time, the Control Plane's control loops will respond to changes in the cluster and work to make the actual state of all the objects in the system match the desired state that you provided.

For example, when the user uses the Kubernetes API to create a Deployment object, he provides a new desired state for the system. The Kubernetes Control Plane records that object creation, and carries out his instructions by starting the required applications and scheduling them to cluster nodes–thus making the cluster's actual state match the desired state.

- **Kubernetes Master** → The Kubernetes master is responsible for maintaining the desired state for your cluster. When the user interacts with Kubernetes, such as by using the kubectl command-line interface, he is communicating with your cluster's Kubernetes master. The "master" refers to a collection of processes managing the cluster state. Typically these processes are all run on a single node in the cluster, and this node is also referred to as the master. The master can also be replicated for availability and redundancy.
- **Kubernetes Nodes** → The nodes in a cluster are the machines (VMs, physical servers, etc) that run your applications and cloud workflows. The Kubernetes master controls each node; user rarely interact with nodes directly.

# Kubernetes | Architecture & design ...

## Overall Components ...

**Master components ...** Master components provide the control plane. Master components make global decisions about the cluster, and do detecting and responding to cluster events (starting up a new pod when a replication controller's 'replicas' field is unsatisfied). Master components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all master components on the same machine, and do not run user containers on this machine.

- **Kube-apiserver** → It is backend server for the K8s control pane that exposes APIs. Kubectl also consumes these APIs.
- **Kube-scheduler** → It plans hosts assignment to newly created pods based on its knowledge on resource stats, requirements, and other aspects.
- **Kube-controller-manager** → It runs and manages different controllers. These controllers usually developed as separate modules but shipped and run in a single process are:
  - Node controller, Replication controller, Endpoint controller, and Service account & token controllers.
- **Cloud-controller-manager** → It runs controllers that interact with the underlying cloud providers. These are:
  - Node controller, Route controller, Service controller, and Volume controller.
- **Etcd** → It provides a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

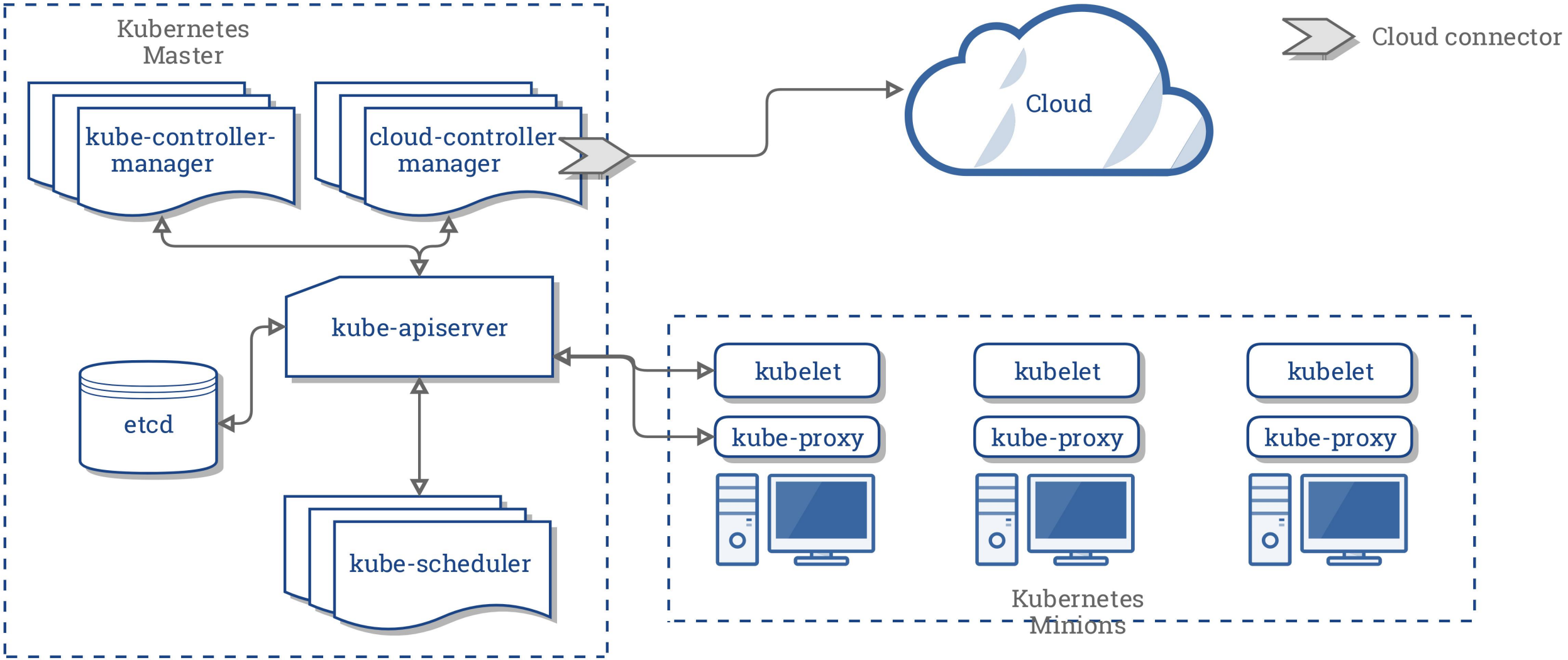
**Node components ...** Node components run on every node or host machines, maintaining running pods and providing the Kubernetes runtime environment.

- **Kubelet** → It is an agent that runs on each node in the cluster. It makes sure that containers are running in a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.
- **Kube-proxy** → It enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.
- **Container Runtime** → It is the software that is responsible for running containers. Kubernetes supports several runtimes: Docker, rkt, runc and any OCI runtime-spec implementation.

**Addons ...** Addons extend the functionality of Kubernetes. They are pods and services that implement cluster features. The pods may be managed by Deployments, ReplicationControllers, and so on. Namespaced addon objects are created in the kube-system namespace. Addons can be four types; Networking, service discoery, control, and legacy (other) addons.

- **Networking & Network Policy Addons**
  - ACI, Calico, Canal, Cilium, CNI-Genie, Contiv, Flannel, Knitter, Multus, NSX-T, Nuage, Romana, Weave Net, ...
- **Service Discovery Addons**
  - CoreDNS
- **Visualization & Control Addons**
  - Dashboard
  - Weave Scope
- **Legacy Addons**

# Kubernetes | (cont.) Architecture & design ...



# Kubernetes | (cont.) Architecture & design …

Kubernetes contains a number of abstractions that represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is doing. These abstractions are represented by objects in the Kubernetes APIs.

| Objects … | |
|---|---|
| Pod | A pod is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster. A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources. Docker is the most common container runtime used in a Kubernetes Pod, but Pods support other container runtimes as well. |
| Service | A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them. The set of Pods targeted by a Service is (usually) determined by a Label Selector. For example, suppose an image-processing backend is running with 3 replicas. Those replicas are fungible - frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that or keep track of the list of backends themselves. The Service abstraction enables this decoupling. |
| Volume | A volume is just a directory, possibly with some data in it, which is accessible to the Containers in a Pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used. It represents a persistent storage in a pod whose lifetime is different and not tied with the pod's containers themself. A volume mounted to the container can provide data storage facility beyond container's lifetime, and hence it can be shared with other containers if required. It has an explicit lifetime - the same as the Pod that encloses it. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously. |
| Namespace | A namespace is a virtual domain that ties up resources in it, and can be use to distinguish resources from one domain to another. A domain can be think of a virtual cluster within the same physical one. They are optional, and are use only if a resource in a cluster is duplicated for another prupose, and in order to address/access duplicate resources individually, each one of them is placed in a separate namespace with their domain resources. A namespace provides a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. They are a way to divide cluster resources between multiple users. |

# Kubernetes | (cont.) Architecture & design …

Kubernetes contains a number of higher-level abstractions called **Controllers**. Controllers build upon the basic objects, and provide additional functionality and convenience features. They include:

| Controllers … | |
|---|---|
| ReplicaSet | A ReplicaSet ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicaSet makes sure that a pod or a homogeneous set of pods is always up and available. Unlike manually created pods, the pods maintained by a ReplicaSet are automatically replaced if they fail, are deleted, or are terminated. ReplicaSet is the next-generation Replication Controller. The only difference between a ReplicaSet and a Replication Controller right now is the selector support. |
| Deployment | A Deployment controller provides declarative updates for Pods and ReplicaSets. The user describes a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate. The user can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments. |
| StatefulSet | Manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these pods. Like a Deployment, a StatefulSet manages pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling. A StatefulSet operates under the same pattern as any other Controller. The user defines the desired state in a StatefulSet object, and the StatefulSet controller makes any necessary updates to get there from the current state. |
| DeamonSet | A DaemonSet ensures that all (or some) Nodes run a copy of a pod. As nodes are added to the cluster, pods are added to them. As nodes are removed from the cluster, those pods are garbage collected. Deleting a DaemonSet will clean up the pods it created. In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types. |
| Job | A job creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the job tracks the successful completions. When a specified number of successful completions is reached, the job itself is complete. Deleting a Job will cleanup the pods it created. A simple case is to create one Job object in order to reliably run one pod to completion. The Job object will start a new pod if the first pod fails or is deleted (for example due to a node hardware failure or a node reboot). A Job can also be used to run multiple pods in parallel. |

# Kubernetes | Purpose …

Kubernetes, being a container orchestration platform, isn't limited to only microservice architecture, or to a certain technology or stack ... but rather it provides implementation and support for all distributed systems concerns for all type of distributed architectures, build using any type of technologies.

**The sole purpose of Kubernetes is to make it easier to organize and schedule one's application across a fleet of machines. At a high level it is an operating system for the application's cluster.**

Basically, it prevent the developer from the concerns of computing resources and their allocation to his applications. It provides generic primitives for health checking and replicating application across these machines, as well as services for wiring the application into micro-services so that each layer in the application is decoupled from other layers, enabling the developer to scale/update/maintain his applications independently.

While it is possible to do many of these things in application layer (like Spring Cloud modules), such solutions tend to be one-off and brittle; it's much better to have separation of concerns, where an orchestration system worries about how to run the application, and the developer worries about the business logic that makes up his application.

# Kubernetes | So, how will Spring Cloud fits in with Kubernetes?

**The answer is simple ... Spring Cloud modules do not work with Kubernetes.**

The Spring Cloud approach is trying to solve every microservice architecture challenge inside the JVM, whereas the Kubernetes approach is trying to make the problem disappear for the developers by solving it at platform level. Spring Cloud is very powerful inside the JVM, and Kubernetes is powerful in managing those JVMs.

Technically, Kubernetes provides its own implementations for all the distributed concerns, and microservice design patterns. And its implementation is more generic in a way that they are not bind with certain language, runtime or framework like Spring Cloud. So, with Kubernetes, the Spring Boot applications donot need to use Spring Cloud modules.

# Kubernetes | Technology mapping to Kubernetes …

**Technology mapping from Spring Cloud → Kubernetes**

| Microservice Concerns | Spring Cloud | Kubernetes |
|---|---|---|
| Configuration management | Spring Cloud Config Server, Consul, Netflix Archaius | Kubernetes ConfigMaps & Secrets |
| Service discovery | Netflix Eureka, Hashicorp's Consul | Kubernetes Service & Ingress Resources |
| Load balancing | Netflix Ribbon | Kubernetes Services |
| API gateway | Netflix Zuul | Kubernetes Service & Ingress Resources |
| Service security | Spring Cloud Security | - |
| Centralized logging | ELK Stack (LogStash) | EFK Stack (Fluentd) |
| Centralized metrics | Spring Cloud Actuator, Admin Server, Netflix Spectator & Atlas | Heapster, Prometheus, Grafana |
| Distributed tracing | Spring Cloud Sleuth, Zipkin | OpenTracing, Zipkin |
| Resilience & fault tolerance | Netflix Hystrix, Turbine & Ribbon | Kubernetes Health Check & resource isolation |
| Auto scaling & self healing | - | Kubernetes Health Check, Self Healing & Autoscaling |
| Packaging, deployment & scheduling | Spring Boot | Docket/Rocket, Kubernetes Schedular & Deployment |
| Job management | Spring Batch | Kubernetes Jobs & Scheduled Jobs |
| Singleton application | Spring Cloud Cluster | Kubernetes Pods |

# Kubernetes | Vs. Spring Cloud frameworks ...

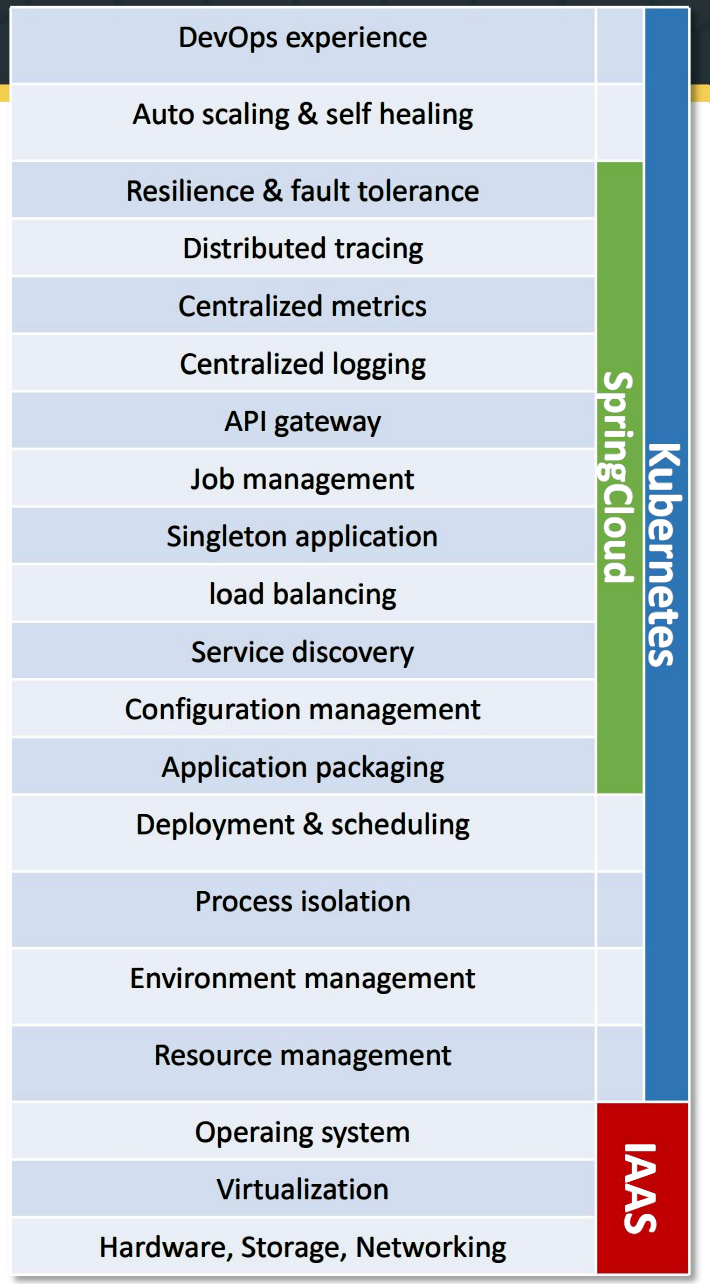
- Spring Cloud has a rich set of well-integrated Java libraries to address all runtime concerns as part of the application stack. As a result, the microservices themselves have libraries and runtime agents to do client-side service discovery, load balancing, configuration update, metrics tracking, etc. Patterns such as singleton clustered services and batch jobs are managed in the JVM too.

- Kubernetes is polyglot, doesn't target only the Java platform, and addresses the distributed computing challenges in a generic way for all languages. It provides services for configuration management, service discovery, load balancing, tracing, metrics, singletons, scheduled jobs on the platform level and outside of the application stack. The application doesn't need any library or agents for client side logic and it can be written in any language.

- In some areas, both platforms rely on similar third-party tools. For example, the ELK and EFK stacks, tracing libraries, etc. Some libraries, such as Hystrix and Spring Boot, are useful equally well on both environments. There are areas where both platforms are complementary and can be combined together to create a more powerful solution ( KubeFlix and Spring Cloud Kubernetes are such examples).

| Spring Cloud | |
|---|---|
| **Strengths** | **Weakness** |
| Unified programming model with Spring framework. | Main feature limited to Java platform. |
| Feature rich collection of Java libraries. | Lots of responsibility for Java devs & the application stack. |
| Well integrated Java libraries. | Doesn't cover full microservice lifecycle. |

| Kubernetes | |
|---|---|
| **Strengths** | **Weakness** |
| Ploygot and generic platform based on containers. | A generic platform with coarse grained primitives. |
| Covers full microservice lifecycle. | Operations focused platform. |
| Cutting edge technology and global community. | Actively developed and rapidly changing. |



# Kubernetes | Spring Cloud support ...

Spring Cloud provides bridging support to Kubernetes through a separate project called **Spring Cloud Kubernetes**.

https://github.com/spring-cloud/spring-cloud-kubernetes

Spring Cloud Kubernetes provide Spring Cloud common interfaces implementations to consume Kubernetes native services. The main objective of the projects provided in this repository is to facilitate the integration of Spring Cloud/Spring Boot applications running inside Kubernetes. The following main modules provides the support Kubernetes services:

| | |
|---|---|
| **Externalized configuration** | The Spring Cloud Kubernetes Config project makes Kubernetes `ConfigMap`s available during application bootstrapping and triggers hot reloading of beans or Spring context when changes are detected on observed `ConfigMap`s. |
| **Sensitive configuration** | Kubernetes has the notion of Secrets for storing sensitive data such as password, OAuth tokens, etc. This project provides integration with Secrets to make secrets accessible by Spring Boot applications. This feature can be explicitly enabled/disabled using the spring.cloud.kubernetes.secrets.enabled property. |
| **Automatic service discovery & registration** | This project provides an implementation of Discovery Client for Kubernetes. This allows you to query Kubernetes endpoints (see services) by name. A service is typically exposed by the Kubernetes API server as a collection of endpoints which represent http, https addresses that a client can access from a Spring Boot application running as a pod. This discovery feature is also used by the Spring Cloud Kubernetes Ribbon project to fetch the list of the endpoints defined for an application to be load balanced. |
| **Service routing & integration with Istio** | The project also provides integration with Istio if it is in use on top of Kubernetes. The Istio awareness module enables the application to interact with Istio APIs enabling it to discover traffic rules, circuit breakers, etc. Making it easy for the Spring Boot applications to consume this data to dynamically configure themselves according the environment. |
| **Leader election** | The project provides integration with the election process in Kubernetes. |
| **Cliend side load balancing through Ribbon** | Spring Cloud client applications calling a microservice should be interested on relying on a client load-balancing feature in order to automatically discover at which endpoint(s) it can reach a given service. This mechanism has been implemented within the project where a Kubernetes client will populate a Ribbon ServerList containing information about such endpoints. |

# Kubernetes | Kubernetes as a service (KAAS) …

As noted earlier, Kubernetes is currently the clear standard for container orchestration tools. It should come as no surprise then that major cloud providers are offering plenty of Kubernetes-as-a-Service offerings:

## Cloud based KAAS solutions …

**Amazon Elastic Container Service for Kubernetes (Amazon EKS)**

https://aws.amazon.com/eks/

Amazon EKS fully abstracts the management, scaling, and security of your Kubernetes cluster, across multiple zones even, so you can focus strictly on your applications and microservices. EKS integrates with popular open source Kubernetes tooling and plenty of AWS tools, including Route 53, AWS Application Load Balancer, and Auto Scaling. The team that manages Amazon EKS are regular contributors to the Kubernetes project.

**Google Cloud Kubernetes Engine**

https://cloud.google.com/kubernetes-engine/

Like Amazon EKS, Kubernetes Engine manages your Kubernetes infrastructure so you don't have to. Google, as the original developer of Kubernetes, has plenty of experiencing running Kubernetes-based containers in production. Kubernetes Engine runs on Google's network and uses routine health checks in high availability configurations and auto scales to meet whatever demand is placed on your applications.

**Azure Kubernetes Service (AKS)**

https://azure.microsoft.com/en-us/services/kubernetes-service/

AKS is Azure's Kubernetes management solution. With AKS, you can secure your clusters with Azure's Active Directory and deploy apps across Azure's massive data center offerings. Azure also provides their own container registry and a provisioning portal. And, as Kubernetes enthusiasts likely already know, Brendan Burns, who co-created Kubernetes, is leading the charge behind Azure's container work.

## Software based KAAS Solution …

**Ranchers …** It is a software solution that provides Kubernetes specific deployment and management.

https://rancher.com/

Rancher not only deploys Kubernetes clusters anywhere, on any provider, but it also unites them under centralized authentication and access control. Because it's agnostic about where the resources run, you can easily bring up clusters in a different provider and deploy applications to all of them. Instead of having several independent Kubernetes deployments, Rancher unifies them as a single, managed Kubernetes Cloud.

RANCHER

# Kubernetes | Conclusion …

**Kubernetes** … has becomes the de-facto solution for highly resilient and scalable deployment. And hence becomes a standard platform for microservice architecture. Means, a system designed with Kubernetes dependencies can easily deployed on local as well as on renowed cloud infrastructure with zero code change.

And … with that, the other platforms, tools, and frameworks (such as Spring Cloud, Netflix OSS) for microservice architecture has being out-dated and replaced.

# Thats it !
# Here I will stop myself.

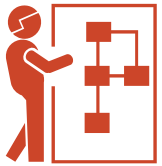
**For journey ahead, do check my future presentations on the topic.**

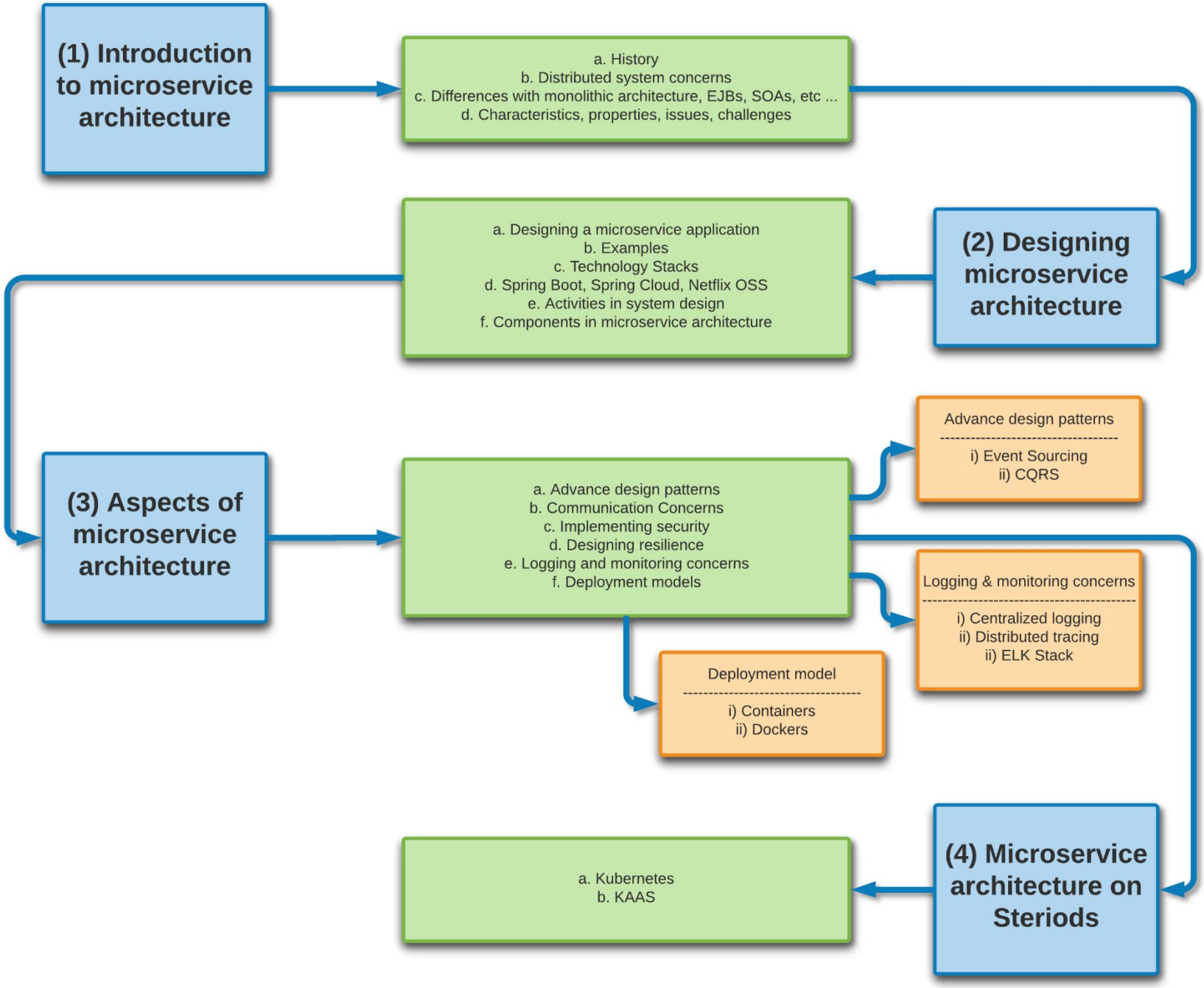
So,

Microservice architecture **PROVIDES** *an strategy for digitizing businesses* **SUCH THAT** the digital version maps the physical one **WITH** design that embrases decoupling and simplicity at the application level **WITH** *natural business boundaries*.

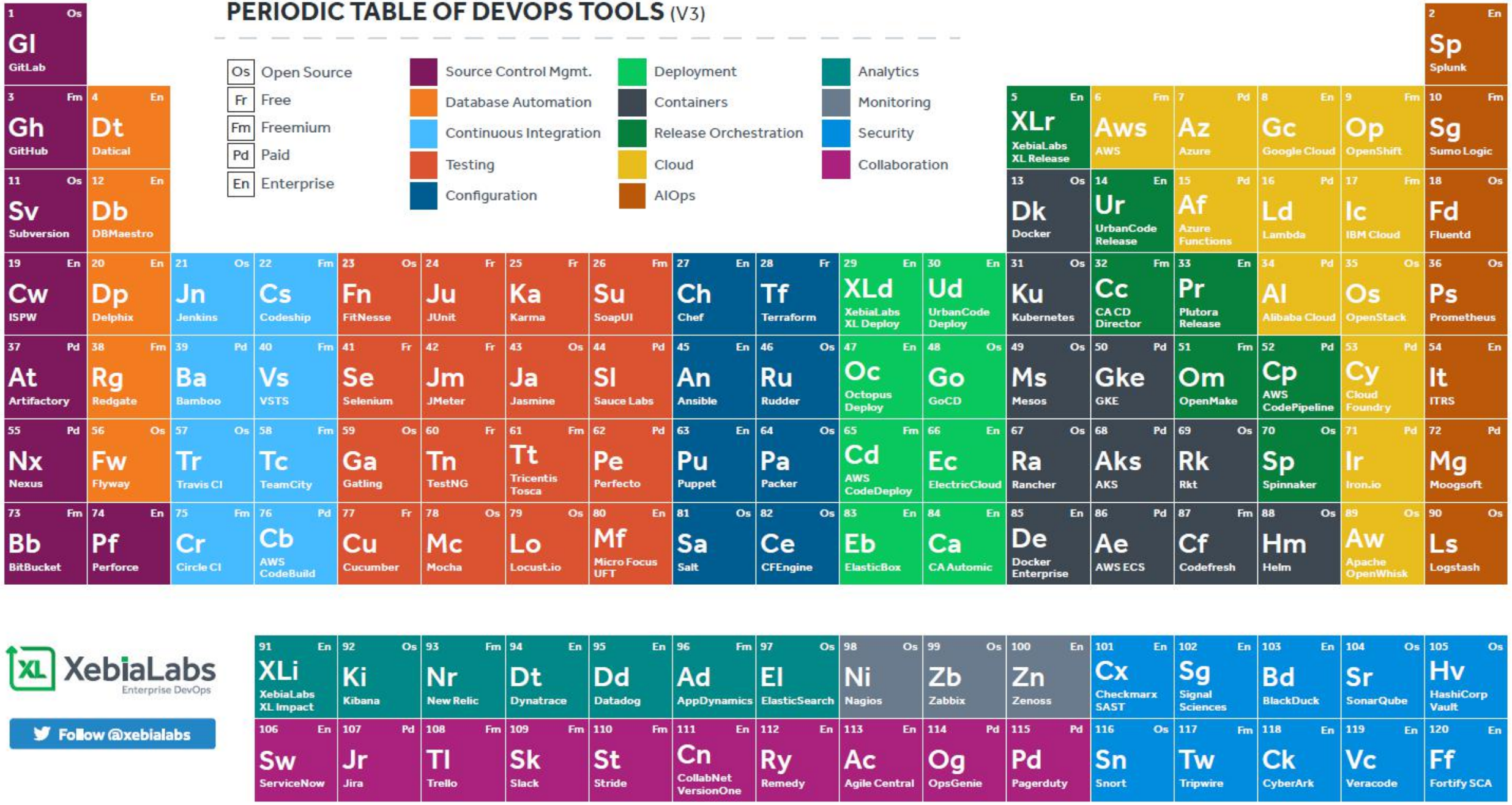
Microservice architecture **IS-A** type of *distributed system design* **THAT** provides solutions for every distributed computing concerns **AS** design patterns **SUCH AS** *communication patterns*, *monitoring patterns*, *resiliency patterns*, *data access patterns*, *deployment patterns*, etc. The implementations of these design patterns are made available **AS** frameworks, tools and platforms such as *SPRING-BOOT*, *SPRING-CLOUD*, *NETFLIX OSS*, *KUBERNETES*, etc.

# OUR JOURNEY ...




(1) Introduction to microservice architecture
a. History
b. Distributed system concerns
c. Differences with monolithic architecture, EJBs, SOAs, etc ...
d. Characteristics, properties, issues, challenges
(2) Designing microservice architecture
a. Designing a microservice application
b. Examples
c. Technology Stacks
d. Spring Boot, Spring Cloud, Netflix OSS
e. Activities in system design
f. Components in microservice architecture
(3) Aspects of microservice architecture
a. Advance design patterns
b. Communication Concerns
c. Implementing security
d. Designing resilience
e. Logging and monitoring concerns
f. Deployment models
Advance design patterns
i) Event Sourcing
ii) CQRS
Logging & monitoring concerns
i) Centralized logging
ii) Distributed tracing
ii) ELK Stack
Deployment model
i) Containers
ii) Dockers
(4) Microservice architecture on Steriods
a. Kubernetes
b. KAAS

# PERIODIC TABLE OF DEVOPS TOOLS (V3)




XebiaLabs
Enterprise DevOps

Follow @xebialabs

https://xebialabs.com/periodic-table-of-devops-tools/

# THANK YOU!

I hope you will have gained a lot of valuable knowledge on the microservice architecture. Please let me know your feedback by leaving comments. And lookout for my next presentation.

In the end, I would like to thank my brain for providing me adequate supply of adrenaline, dopamine, noradrenaline, serotonin, and oxytocin for keeping me motivated and over the edge. Without them, I wouldn't have completed this deck.