Microservices

what are micro services?

micro services also known as microservice architecture- is an architecture style that structures an application of collection of service that are

- highly maintainable and testable

- loosely coupled

- etc...

- A small code to perform big task.

what is monolithic?

-> All service in one bundle. (tightly couple)

Characteristics of a Microservice Architecture

1.Componentization via Services

2.Organized around Business Capabilities

3.Products not Projects ("you build, you run it")

4.Smart endpoints and dumb pipes

5.Decentralized Governance

6.Decentralized Data Management

7.Infrastructure Automation

8.Design for failure

Intelligence in the endpoints

<--------------------------🡪

Q) What are smart End Points?

Ans) Smart end points are HTTP which is used in the rest which handles the user request through the URL.

-GET, POST, DELETE, PUT, ETC...

Q) What is Dump pips?

Ans) dumb pipes used to get messages from one endpoint to another endpoint. (messages means kind of a request).

->message Architecture

->Point-To-Point channel. - (Queue)

->Publish-Subscribe channel. - (topic)

Principles of microservices

<---------------------------->

--> These principles are a must have when designed and developing microservices.

->single responsibility per service.

->Microservices are autonomous.

Single responsibility

<--------------------->

* single responsibility principles are one of the principles is defined of the SOLID design pattern.
* It states that a util should only have one responsibility

customer

multiple responsibility Monolithic App

Monolithic App

customer

product

Order

Order

product

Single responsibility microservices

**Microservices are autonomous.**

* Autonomous microservices are self-contained software units that can be deployed independently and encapsulate a specific business capability. They are designed to operate independently of other services, which reduces communication overhead and dependencies.

**Microservices are lightweight**

Why microservices are lightweight.?

* Well-designed microservices are aligned to a single business capability, so they perform only one function. As a result, one of the common characteristics we see in most of the implementations are microservices with smaller footprints.
* When selecting supporting technologies, such as web containers, we will have to ensure that they are also lightweight so that the overall footprint remains manageable. For example, Jetty or Tomcat are better choices as application containers for microservices compared to more complex traditional application servers such as WebLogic or WebSphere.
* Container technologies such as Docker also help us keep the infrastructure footprint as minimal as possible compared to hypervisors such as VMWare or Hyper-V.

**Stereotype**

@Component (Parent component)

| @Controller, @Service, @Repository, @RestController, | (Child annotations of @Components).

To give mapping for a Controller we have to use @RequestMapping annotation for the controller.

* GetMapping
* PostMapping
* DeleteMapping
* putMapping

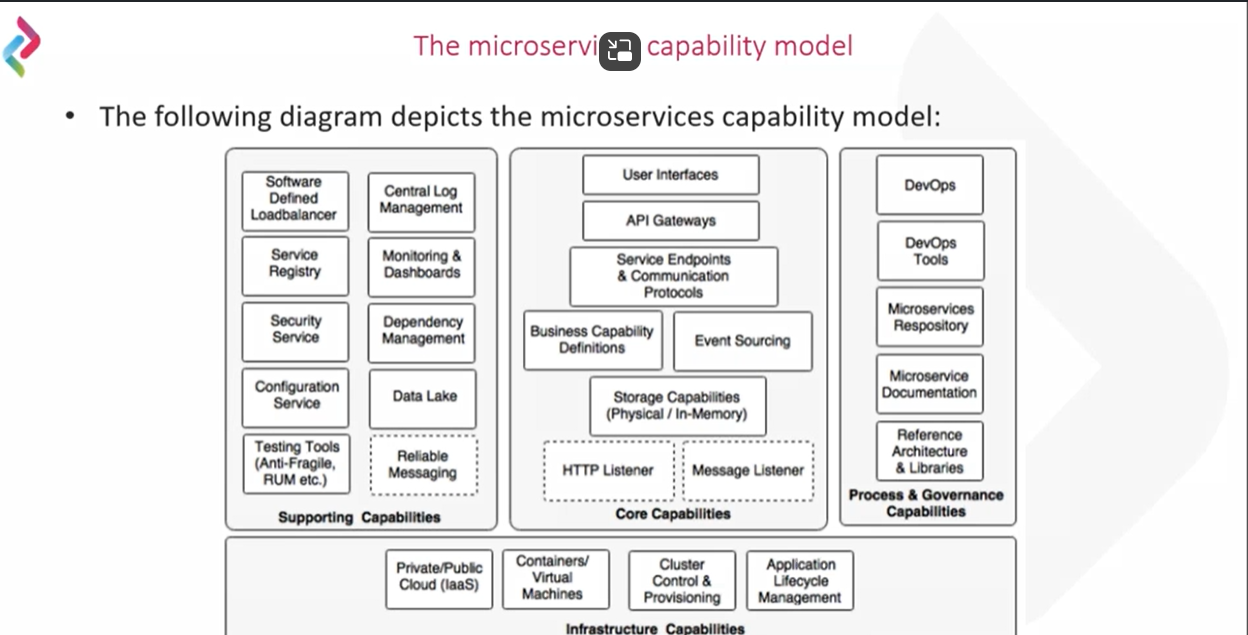
These are the Child annotation of the @Request mapping and we call it as smart end points.

STATUA CODES

1. When the Record is created the status code is -> 201
2. When the record is updated the status code is -> 200
3. When the record is deleted the status code is - > NO\_CONTENT

Producer and consumers.

* Producer is used sending data format where it is Json, xml, etc...
* Consumers is used to reserve data in fixed format where it is Json, xml, etc



**The capability model is broadly classified in to four areas:**

**Core capabilities**: These are part of the microservices themselves

**The core capabilities are explained as follows:**

* Service listeners (HTTP/messaging)
* Storage capability
* Business capability definition
* Event sourcing
* Service endpoints and communication protocols
* API gateway
* User interfaces

**Supporting capabilities**: These are software solutions supporting core microservice implementations

**The Supporting capabilities are explained as follows:**

* Software defined Load Balancer
* Central log management
* Service registry
* Security service
* Service configuration
* Testing tools (anti-fragile, RUM and so on)
* Monitoring and dashboards
* Dependency and Cl management
* Reliable Messaging

**Infrastructure capabilities**: These are infrastructure level expectations for a successful microservices implementation

**The infrastructure capabilities are explained as follows:**

* Cloud
* Containers or virtual machines
* Cluster control and provisioning
* Application lifecycle management

**Governance capabilities**: These are more of process, people, and reference information

**The Process and governance capabilities are explained as follows:**

* DevOps
* DevOps tools
* Microservices repository
* Microservices documentation
* Reference architecture and libraries

**Data Source:** - It is a pool of connections.

Name of the default profile in spring boot is: **default**

**Spring Boot Actuator**

Spring Boot Actuator is a powerful tool provided by Spring Boot that helps in monitoring and managing applications in production. It provides a set of endpoints that allow you to access information about the application’s health, metrics, environment, configurations, and more.

**Key Features of Spring Boot Actuator:**

1. **Health Checks**: Provides details about the application's health, including the status of various components like databases, caches, messaging systems, etc.
2. **Metrics**: Exposes a variety of metrics related to the JVM, memory usage, garbage collection, and custom application metrics.
3. **Environment Information**: Displays details about the current environment, including properties, configuration files, and environment variables.
4. **HTTP Trace**: Captures the last few HTTP requests and responses, which can be useful for debugging and performance monitoring.
5. **Audit Events**: Tracks security-related events such as authentication successes and failures.

**Common Actuator Endpoints:**

* **/actuator/health**: Provides the health status of the application.
* **/actuator/metrics**: Shows various application and system metrics.
* **/actuator/env**: Displays environment properties.
* **/actuator/beans**: Lists all Spring Beans in the application context.

**Application. Properties**

* Management.endpoints.web.exposure.include=\* (this line will show all activators in the application)

If we want show the values in env object

* Management.endpoint.env.show-values=always (it will show all the end points value when you load the actuator/env)
* **Database connection in application properties**
* Spring.datasourse.driver-class-name= (for driver class and database name) SQL(**com.mysql.cj.jdbc.Driver**), H2(**org.h2.Driver**)
* Spring.datasource.url= URL connection
  + **SQL: - (jdbc:mysql://localhost:3306/stocks?createDatabaseIfNotExist=true) H2: -(jdbc:h2:mem:databasename)**
* Spring.datasourse.username= (for username)
* Spring.datasourse.password= (for password)

**H2 DATABASE CONSOLE**

* Spring.h2.console.enabled =true(it enable the h2 database)
* Spring.h2.console.path=/h2-console(it enables the path for h2 console)

SPRING CLOUD

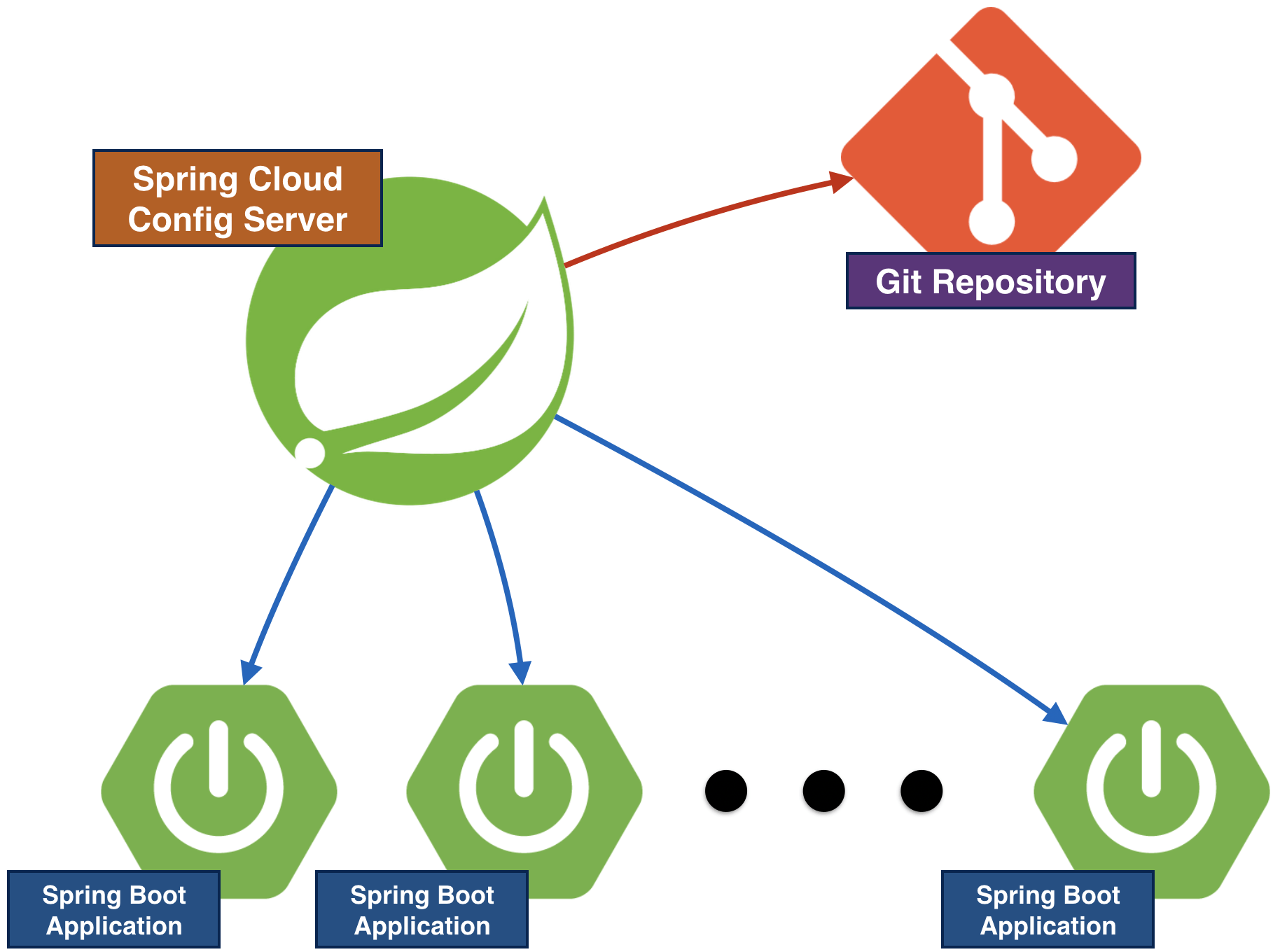
****

**1.Distributed configuration OR Spring cloud config**

**To set profile: -**

In VM level arguments [-Dspring.profiles.active=dev] **Or** [-Dspring.profiles.active=prod]

**12 Factors for microservices. (** [**The Twelve-Factor App (12factor.net)**](https://12factor.net/)**)**



1. Codebase.
   1. One Codebase tracker in revision control, many deploys (that means message services in one deploy, production and code in one deploy, SMS in one deploy, etc...) ([check](https://12factor.net/codebase))
      1. Steps To implementation of Spring config cloud.
         1. Create a new repo and upload all property files into repo
         2. Create a new spring cloud project and add the dependency of spring cloud config**(spring-cloud-config-server)**.
         3. Annotated the main class with **@EnableConfigServer**
         4. In the application properties configure the upload files in git repository

In application properties of cloud project

* + - * 1. Spring.cloud.config.server.git.urI= **git URI**
        2. Spring.cloud.config.server.git.username = (**If the git hub is private account**).
        3. Spring.cloud.config.server.git.password= (**If the git hub is private account**).
        4. Spring.cloud.config.server.git.clone-on-start=true (**If we want to get files from the git repository**).
        5. Spring.cloud.config.server.git.basedir=file://${user.dir}/foldername (**By Writing this line after cloning the files will be moved to this folder if the folder is not existed it will create a new file**).
        6. To test the properties file which is clone: - [**Http://localhost:potnumber/{name}/{profile}/{label}**](Http://localhost:potnumber/%7bname%7d/%7bprofile%7d/%7blabel%7d) **(name=application, profile = file name, label = git branch name)**
    1. Steps To implementation of Spring config clint.
       1. Create a spring project and add the dependency (**spring-cloud-stator-config**)
       2. Go to application set the configuration for cloud
          1. Server.port=8080
          2. Spring.config.import=optional:configserver:http://localhost:8888 (It will connect to the server).
          3. Spring.cloud.config.label= **Git hub branch name (ex: -main, master). (**if we want to set the branch name at application.proerties**)**
          4. Spring.application.name = **application name (representing file name ex: - application). (**if we want to set the name name at application.proerties**)**
          5. We can set the profile name in Eclipse configuration

-Dspring.profile.active=profile name (dev)

-Dspring.profile.active=profile name(prod)

**Changing the application name for the spring cloud config clint.**

1. Go to git hub change the file name to some name.
2. Go to application.properties in client, Change the **spring.application.name=file name**

**@RefreshScope: -** Will be refresh the bean when anything changed in entered the application

* When we apply **server.port=0** application. It will pick-up random port which is unused in the system and run on the port number.

**SPRING CLOUD BUS**

**Spring Cloud Bus: -**

Spring Cloud Bus is a lightweight message broker that connects nodes in a distributed system and can be used to broadcast state changes and management instructions

* Spring Cloud Bus links nodes of a distributed system with a lightweight message broker.
* This can then be used to broadcast state changes (e.g., configuration changes) or other management instructions.
* AMQP and Kafka broker implementations are included with the project.
* Alternatively, any Spring Cloud Stream binder found on the class-path will work out of the box as a transport.

**For install and run the rabbit-MQ**

# latest RabbitMQ 3.13

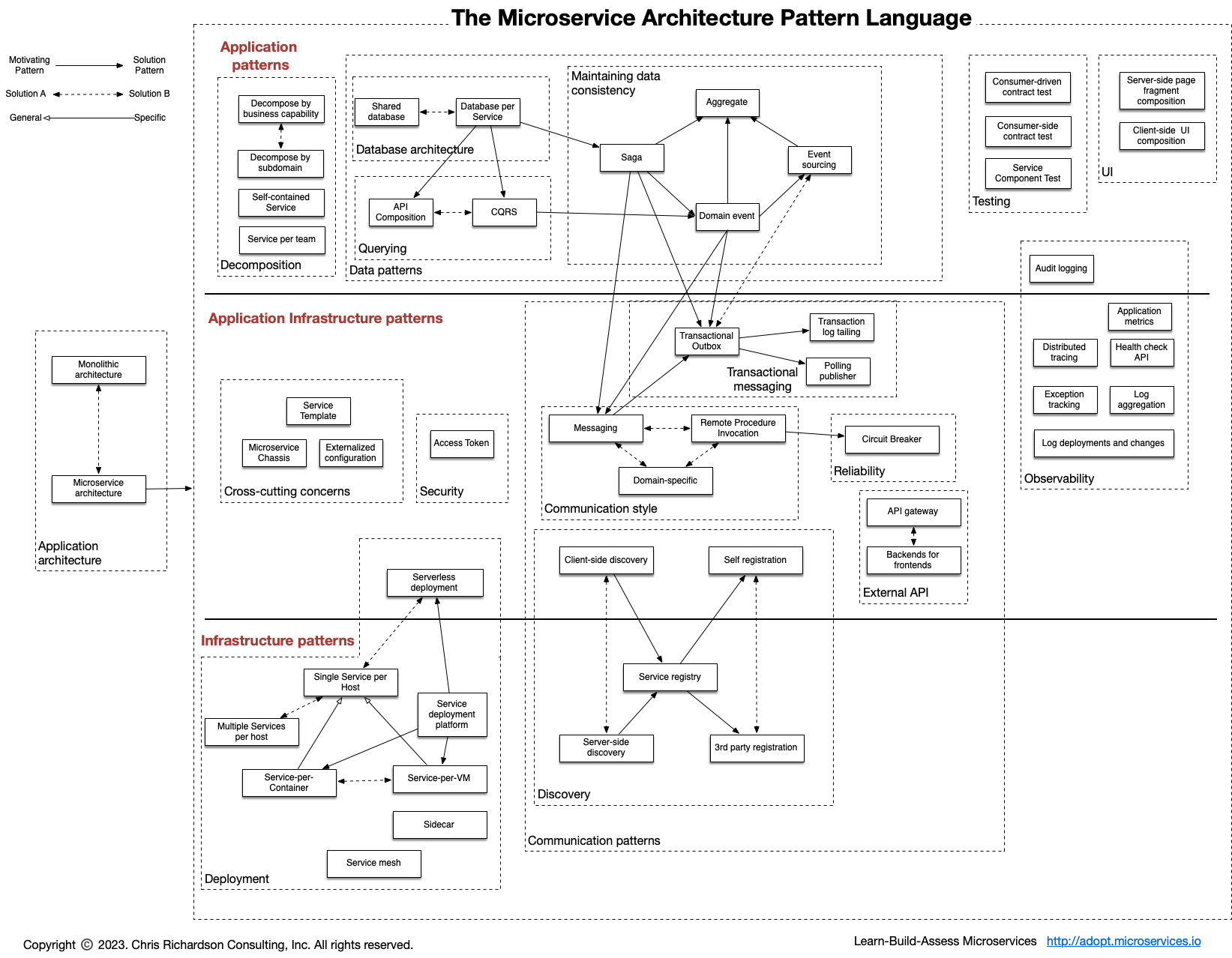
* **docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.13-management**
* radditMQ dashboard running on the pot number **15672**

**Connection for rabbit-MQ in spring cloud**

* spring.rabbitmq.host=localhost
* spring.rabbitmq.port=5672
* spring.rabbitmq.username=guest
* spring.rabbitmq.password=guest
* management.endpoint.web.exposure.include=busrefresh (It Enable the busrefresh actuator)

**2. Service Registration and Discovery**

**microservices patterns: -** [**A pattern language for microservices**](https://microservices.io/patterns/)



Patterns of microservices are characterised into three types

1. application pattern
2. application infrastructure patterns
3. infrastructure pattern

**SPRING CLOUD NETFLEX**

**Spring Cloud Netflix**

* 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.
* The patterns provided include Service Discovery (Eureka), Circuit Breaker.
* (Hystrix), Intelligent Routing (Zuul) and Client-Side Load Balancing (Ribbon).

**Problem: -** When the instance is started the port number and the IP’s is changing every time. For this we will not register manually every time.

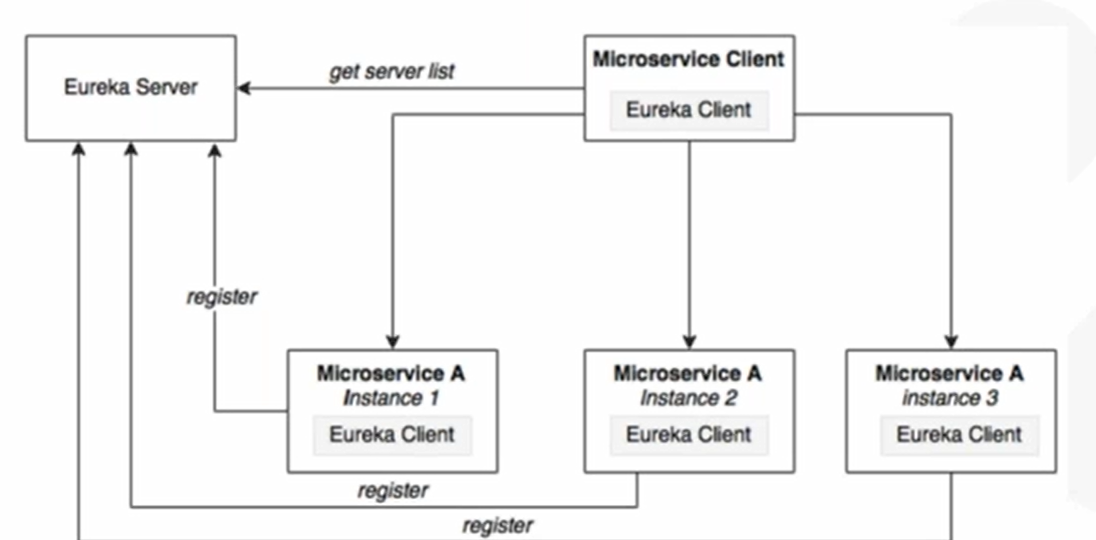
**Solution: -**

**Understanding Dynamic Service Registration and Discovery**

* **Dynamic registration** is primarily from the service provider's point of view. With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the service registry. The registry always keeps up-to-date information of the services available, as well as their metadata.
* **Dynamic discovery** is applicable from the service consumer's point of view. Dynamic discovery is where clients look for the service registry to get the current state of the services topology, and then invoke the services accordingly. In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.
* There are several options available for dynamic service registration and discovery.
* Netflix Eureka, Zookeeper, and Consul are available as part of Spring Cloud.

**Understanding Eureka**

* Spring Cloud Eureka also comes from Netflix OSS. The Spring Cloud project provides a Spring-friendly declarative approach for integrating Eureka with Spring-based applications.
* Eureka is primarily used for self-registration, dynamic discovery, and load balancing.

****

**Setting up Eureka server: -**

1. Create a new spring project with adding the dependency (Eureka server).

**<dependency>**

**<groupId>org.springframework.cloud</groupId>**

**<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>**

**</dependency>**

1. Create an application properties.
   1. server.port=8761
   2. eureka.client.register-with-eureka=false (**If we want to registry eureka server in eureka {no->false / yes->true}** )
   3. eureka.client.fetch-registry=false. (**If we want fetch the registry** **{no->false / yes->true}**).
2. Add @EnableEurekaServer in application.java
   1. @EnableEurekaServer
   2. @SpringBootApplication
   3. Public class EurekaServer{}.
3. We are now ready to start the Eureka server. Once the application is started, Open the <http://localhost:8761> in the browser to see the Eureka dashboard or console.

**Setting up dynamic registration: -**

1. Add the dependency in the pom.xml of application client

**<dependency>**

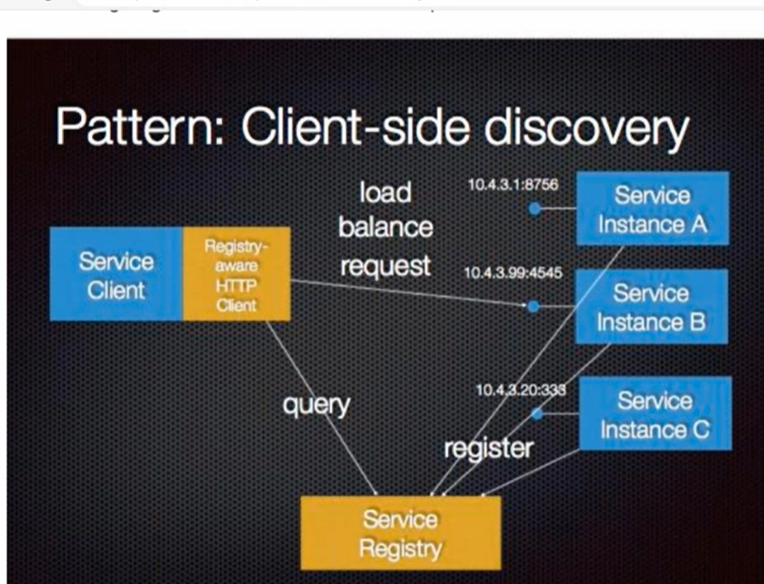
**<groupId>org.springframework.cloud</groupId>**

**<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>**

**</dependency>**

1. Add eureka server configuration in application properties
   1. Eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka (**Eureka server URL.**).
   2. Eureka.instance.preferIpAddress=true (**It will save the IP address of the instance in the eureka server.**).
2. Add the annotation to the main class @EnableDiscoveryClient.

**Setting up dynamic discovery.**



**Ribbon for load balancing**

* Netflix ribbon is an inter process communication (IP) cloud library.
* Ribbon primarily provides client-side load balancing algorithms.

Microservice A

Instance 1

Service client

(With ribbon)

()

Microservice B

Instance 3

Microservice C

Instance 3

**Client-side vs server-side load balancing**

* The multiple instances of the same microservice are run on different computers for high reliability and availability.
* Server-side load balancing is distributing the incoming requests towards multiple instances of the service.
* Client-side load balancing is distributing the outgoing request from the client itself.