Mphasis
The Next Applied

# Java Microservices – S1

Manpreet Singh Bindra
Manager

# Do’s and Don’t

- Login to GTW session on Time
- Login with your Mphasis Email ID only
- Use the question window for asking any queries

# Welcome

1. Skill - Proficiency Introduction

2. About Me - Introduction

3. Walkthrough the Skill on TalentNext

4. About Peer Learning

# About Peer Learning Platform

Hi folks! I am preparing for the AWS Certified Developer – Associate Certification. It has been a wonderful experience, so far.

How I wish there was a forum where I could post questions, follow discussions, seek practical insights.

You are in luck, my friend! Say hello to "Peer Learning" feature on Talent Next! It's a collection of discussion forums on specific skills.

You can follow ongoing discussions, contribute to it by sharing your learnings and create new discussion threads to seek clarification. All these are bound to add to your overall learning experience.

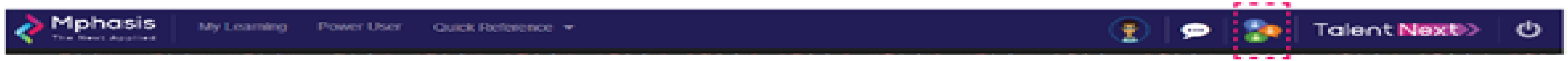
Yes indeed! Peer Learning comes handy to get quick responses to your queries as a wide pool of subject matter experts interact actively

This is awesome! Thank you, folks

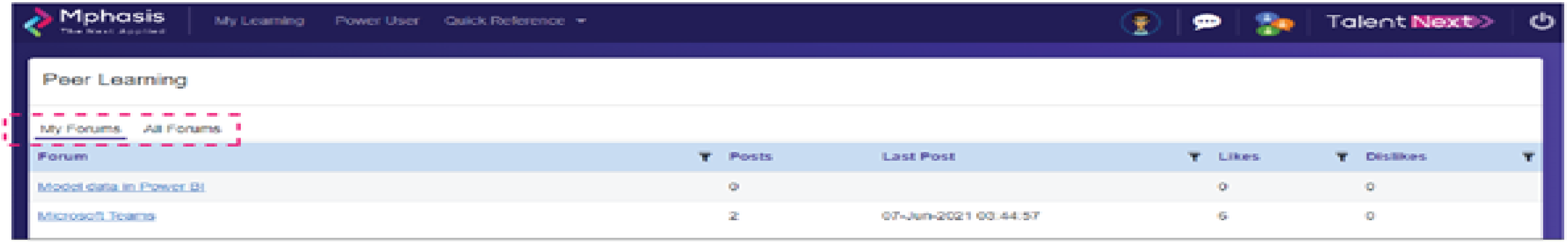
# Where to find Peer Learning Platform

A quick way to get started:

Step 1: Click on the Peer Learning Icon at the top right of the menu bar



Step 2: Select the forum you want to post in from the list

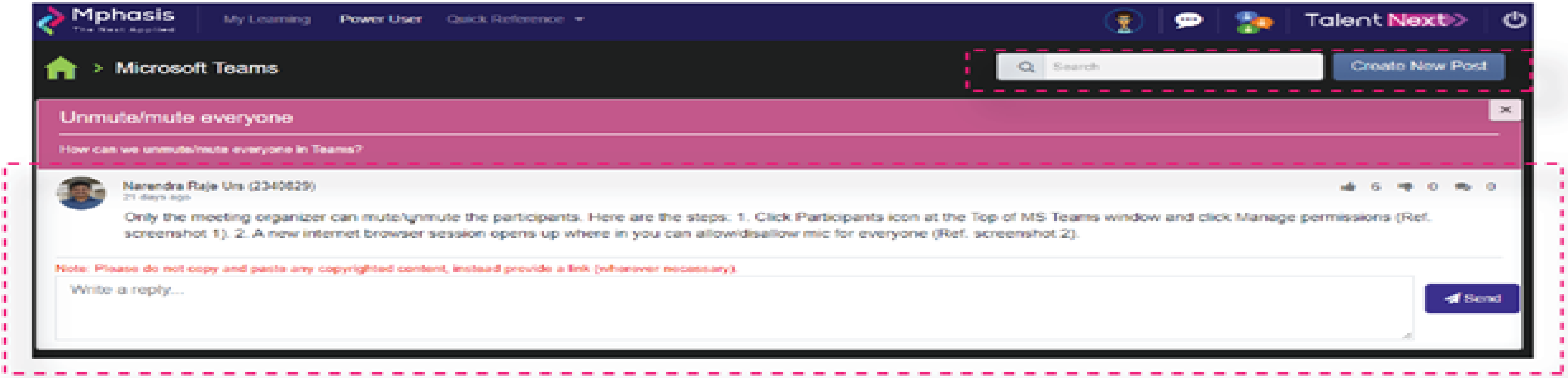


Peer Learning

My Forums All Forums

| Forum | Posts | Last Post | Likes | Dislikes |
|---|---|---|---|---|
| Model data in Power BI | 0 | | 0 | 0 |
| Microsoft Teams | 2 | 07-Jun-2021 03:44:57 | 6 | 0 |

Step 3: Use the “Create New Post” to create a new discussion thread



Microsoft Teams

Unmute/mute everyone

How can we unmute/mute everyone in Teams?

Narendra Raja Urs (2340829)
21 days ago

6 0 0

Only the meeting organizer can mute/unmute the participants. Here are the steps: 1. Click Participants icon at the Top of MS Teams window and click Manage permissions (Ref. screenshot 1). 2. A new internet browser session opens up where in you can allow/disallow mic for everyone (Ref. screenshot 2).

Note: Please do not copy and paste any copyrighted content, instead provide a link (wherever necessary).

Write a reply...

# Overall Agenda

- Use Spring Boot to build standalone web applications and RESTful services
- Understand the Configuration techniques that Spring Boot Provides
- Build Spring boot based Microservices for JSON and XML data exchange
- Monitor services using the Actuator
- Understand the major components of Netflix OSS
- Register services with a Eureka Service
- Implement “client” load balancing with Ribbon to Eureka managed Services
- Isolating from failures with Hystrix
- Filter requests to your Microservices using Zuul
- Define Feign clients to your services
- Scaling Microservices with Spring Cloud

# Day - 1

# Day – 1 Agenda

- What Is Monolithic Architecture?
- Concerns With the Monolith
- The Microservice architecture
- Characteristics of a Microservice Architecture
- Principles of microservices
- Business demand as a catalyst for Microservices
- Technology as a catalyst for the microservices evolution
- Building microservices with spring boot
- The microservices capability model

# Day - 1

# Introduction to Microservices

# What are Microservices?

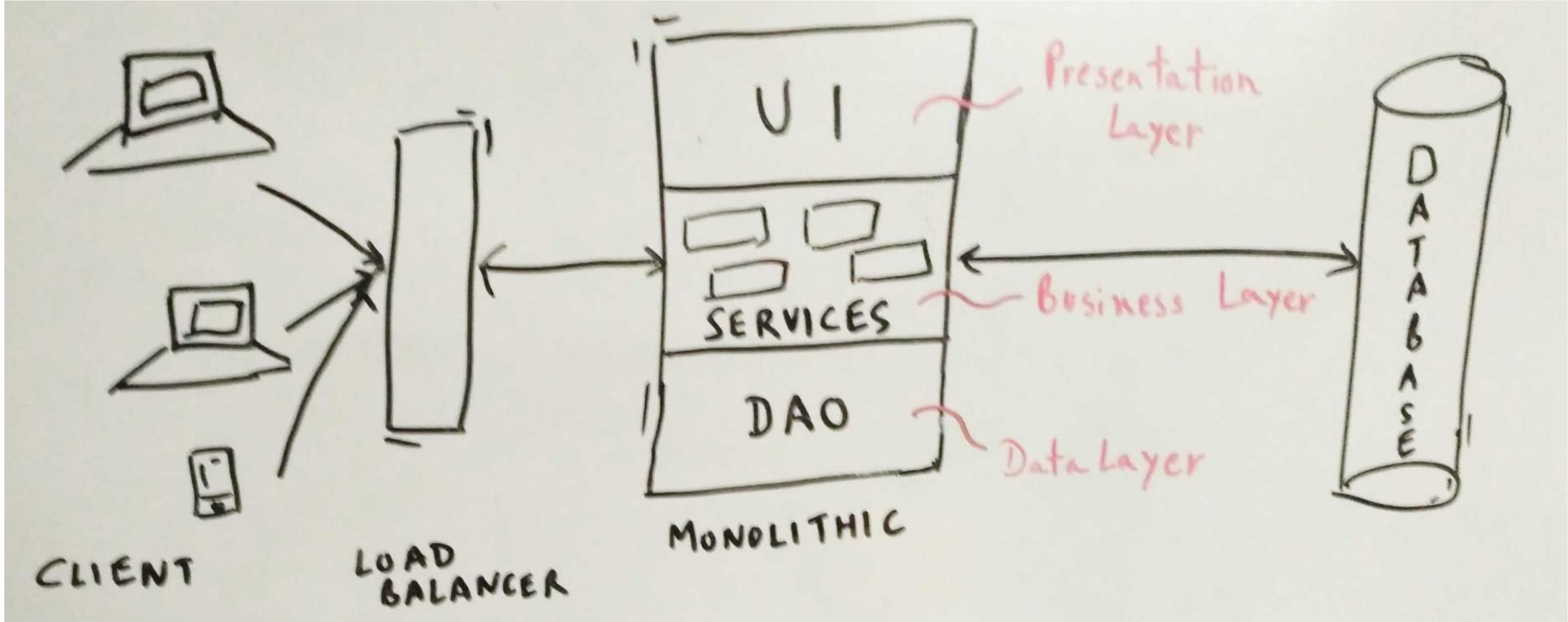
- Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are
  - Highly maintainable and testable
  - Loosely coupled
  - Independently deployable services
  - Organized around business capabilities
  - Owned by a small team

# Need for Microservices

- To understand the need for microservices, we need to understand problems with our typical 3-tier monolithic architecture.

# What Is Monolithic Architecture?

- Monolithic means composed all in one piece. A monolithic application is one which is self-contained. All components of the application must be present in order for the code to work.



# Concerns With the Monolith

- The large monolithic code base
- Overloaded IDE
- Overloaded web container
- Continuous deployment is difficult
- Scaling the application can be difficult
- Obstacle to scaling development
- Requires a long-term commitment to a technology stack

# The Microservice architecture

- The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.
- While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

# Characteristics of a Microservice Architecture

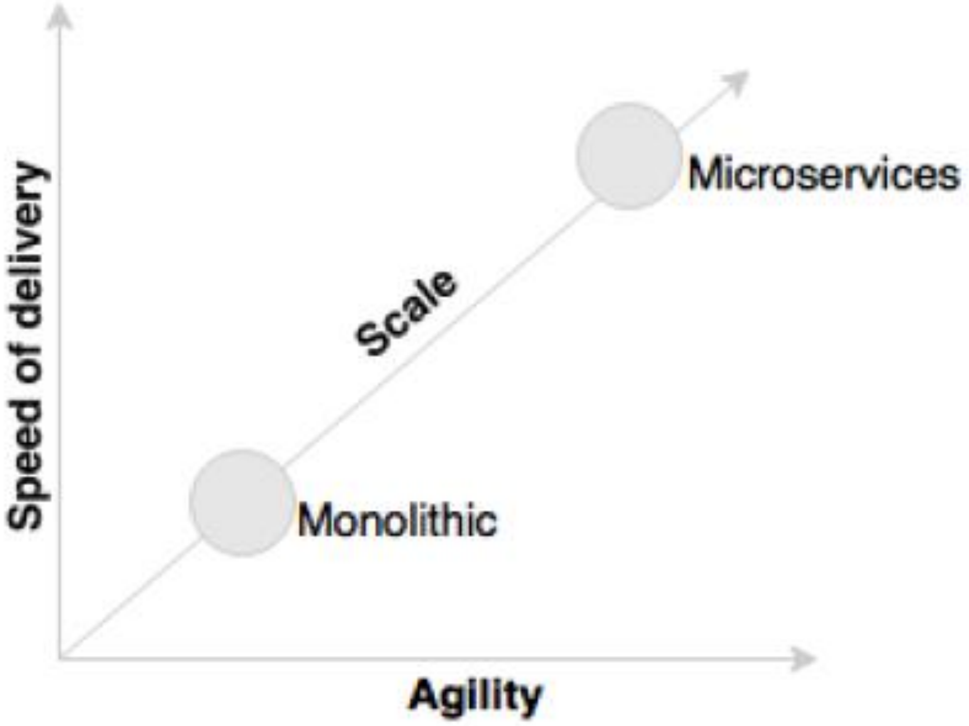
- Componentization via Services
- Organized around Business Capabilities
- Products not Projects ("you build, you run it")
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure

# Business demand as a catalyst for Microservices

- In this era of digital transformation, enterprises increasingly adopt technologies as one of the key enablers for radically increasing their revenue and customer base.
- Enterprises primarily use social media, mobile, cloud, big data, and Internet of Things as vehicles to achieve the disruptive innovations. Using these technologies, enterprises find new ways to quickly penetrate the market, which severely pose challenges to the traditional IT delivery mechanisms.

# Business demand as a catalyst for Microservices

- The following graph shows the state of traditional development and microservices against the new enterprise challenges such as agility, speed of delivery, and scale.



# Technology as a catalyst for the microservices evolution

- Emerging technologies have also made us rethink the way we build software systems.
- The emergence of HTML 5 and CSS3 and the advancement of mobile applications repositioned user interfaces. Client-side JavaScript frameworks such as Angular, Ember, React, Backbone, and so on are immensely popular due to their client-side rendering and responsive designs.
- With cloud adoptions steamed into the mainstream, **Platform as a Services (PaaS) providers** such as Pivotal CF, AWS, Salesforce.com, IBMs Bluemix, RedHat OpenShift, and so on made us rethink the way we build middleware components.
- The container revolution created by **Docker** radically influenced the infrastructure space. These days, an infrastructure is treated as a commodity service.

# Technology as a catalyst for the microservices evolution

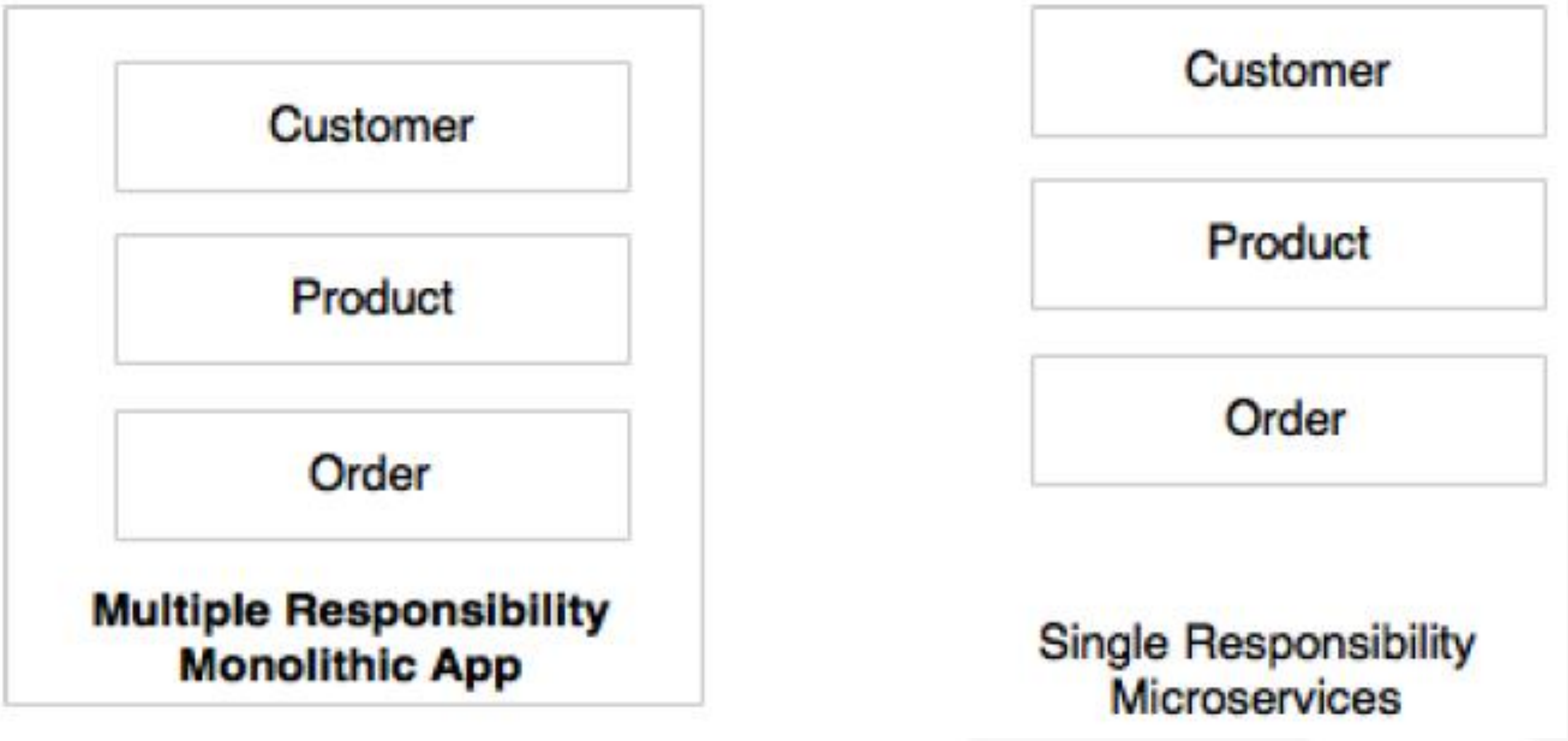
- The integration landscape has also changed with **Integration Platform as a Service (iPaaS)**, which is emerging. Platforms such as Dell Boomi, Informatica, MuleSoft, and so on are examples of iPaaS.
- NoSQLs have revolutionized the databases space. A few years ago, we had only a few popular databases, all based on relational data modeling principles. We have a long list of databases today: Hadoop, Cassandra, CouchDB, and Neo 4j to name a few.

# Principles of microservices

- These principles are a "must have" when designing and developing microservices.
  - Single responsibility per service
  - Microservices are autonomous

# Single responsibility per service

- The single responsibility principle is one of the principles defined as part of the SOLID design pattern.
- It states that a unit should only have one responsibility.



Customer

Product

Order

**Multiple Responsibility Monolithic App**

Customer

Product

Order

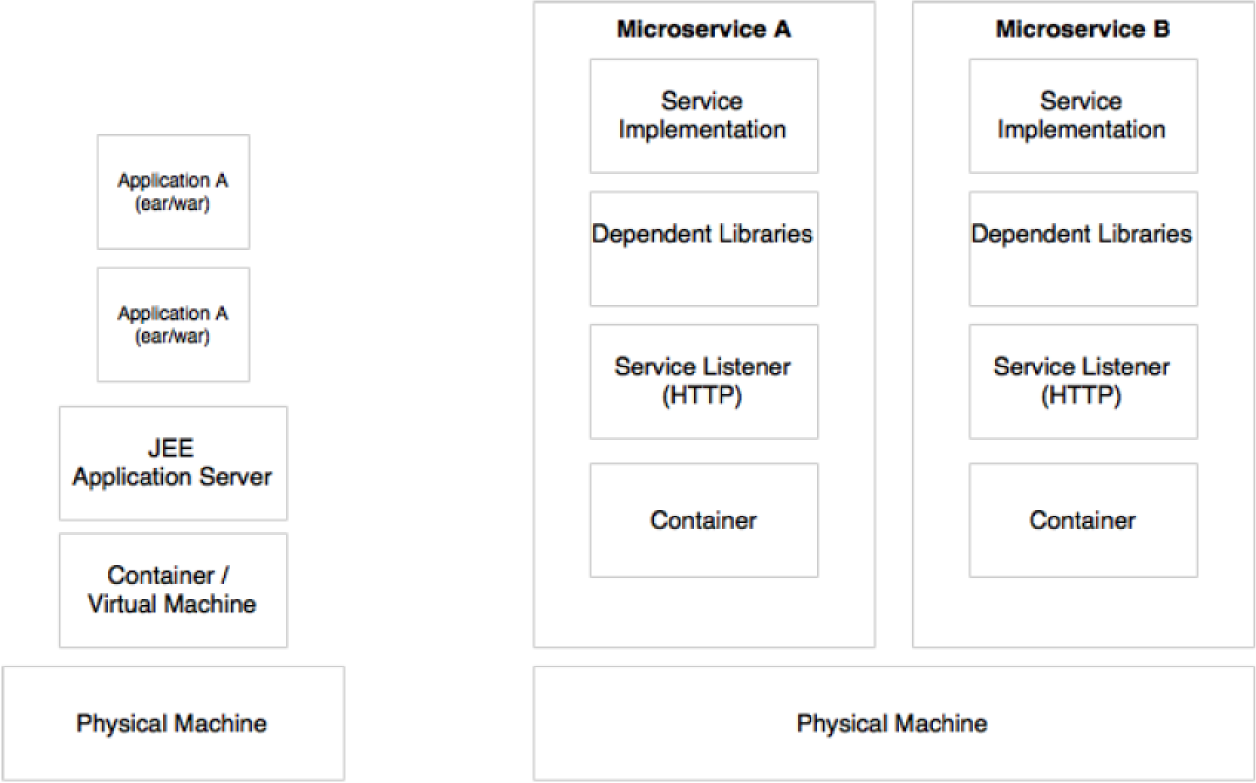
Single Responsibility Microservices

# Microservices are autonomous

- Microservices are self-contained, independently deployable, and autonomous services that take full responsibility of a business capability and its execution.
- They bundle all dependencies, including library dependencies, and execution environments such as web servers and containers or virtual machines that abstract physical resources.

# Major differences between Microservices and SOA

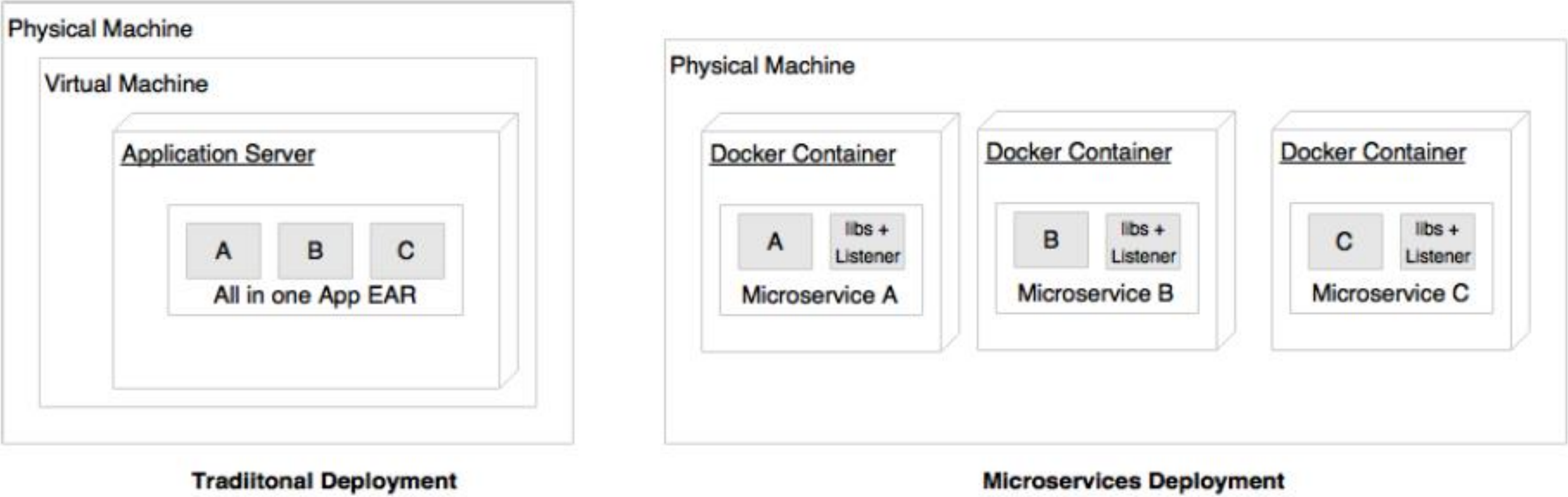
- One of the major differences between microservices and SOA is in their level of autonomy. While most SOA implementations provide service-level abstraction, microservices go further and abstract the realization and execution environment.



# 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.

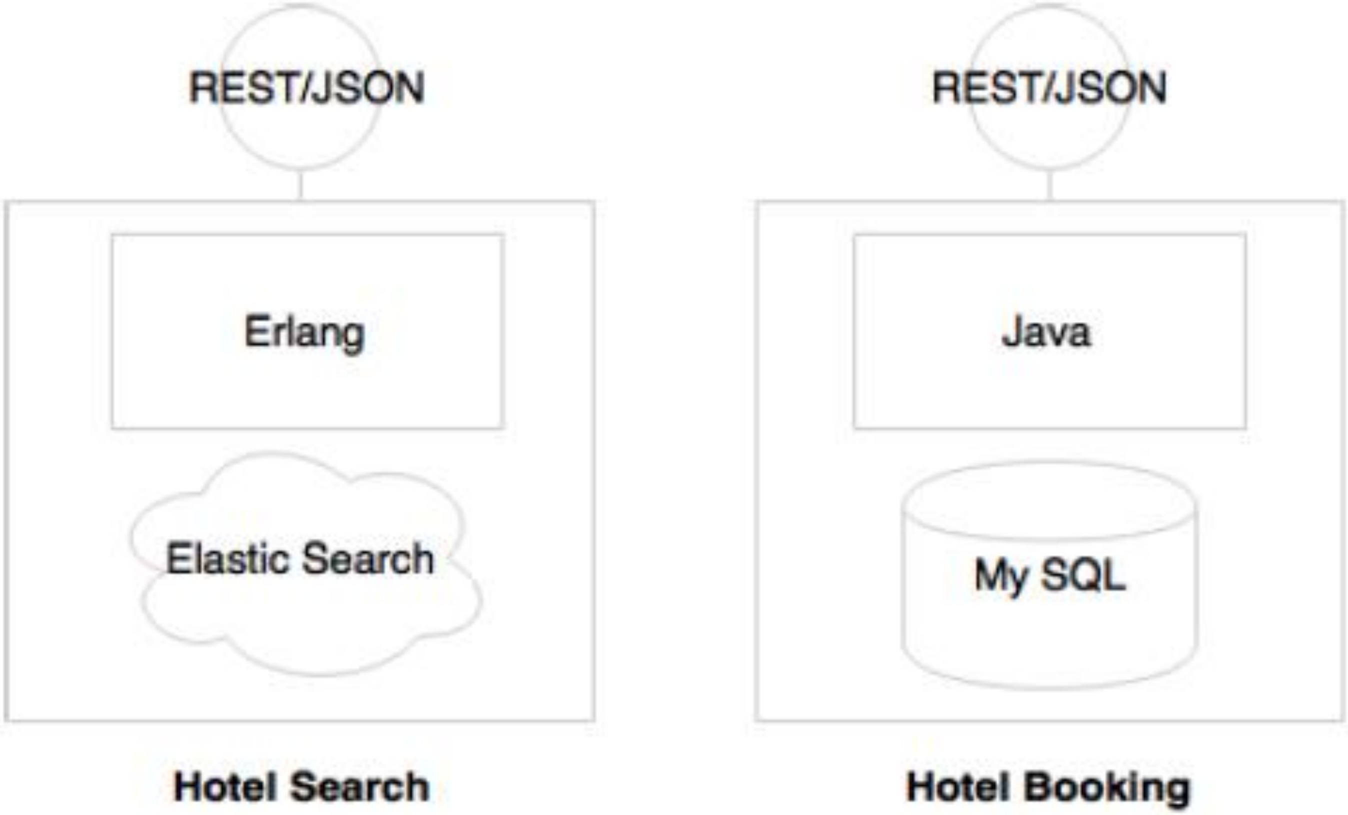
# Microservices are lightweight



# Microservices with polyglot architecture

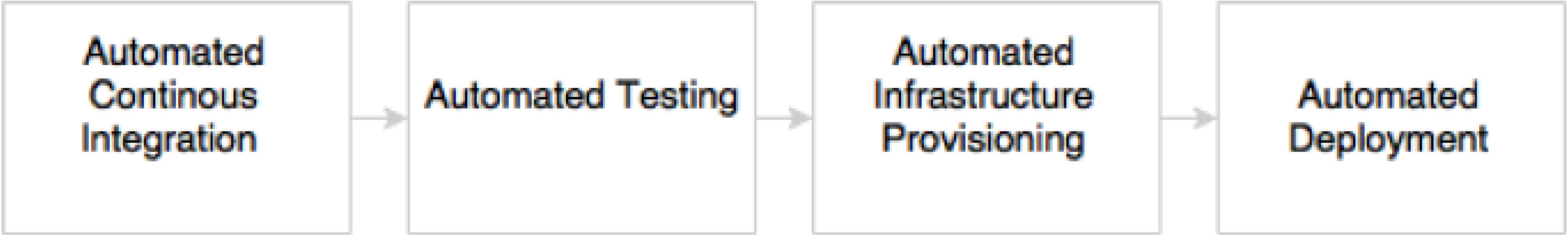
- As microservices are autonomous and abstract everything behind service APIs, it is possible to have different architectures for different microservices.
- A few common characteristics that we see in microservices implementations are:
  - Different services use **different versions** of the same technologies. One microservice may be written on Java 1.7, and another one could be on Java 1.8.
  - **Different languages** are used to develop different microservices, such as one microservice is developed in Java and another one in Scala.
  - **Different architectures** are used, such as one microservice using the Redis cache to serve data, while another microservice could use MySQL as a persistent data store.

# Microservices with polyglot architecture



# Automation in a microservices environment

- Most of the microservices implementations are automated to a maximum from development to production.



Automated Continous Integration

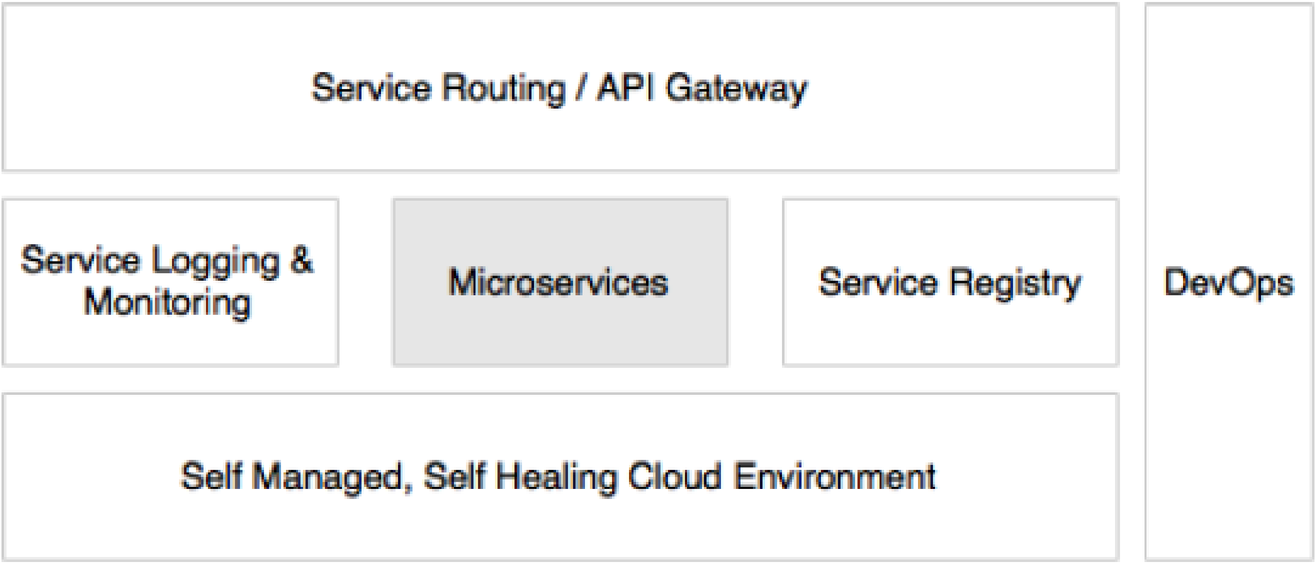
Automated Testing

Automated Infrastructure Provisioning

Automated Deployment

# Microservices with a supporting ecosystem

- Most of the large-scale microservices implementations have a supporting ecosystem in place.
- The ecosystem capabilities include
  - DevOps processes
  - Centralized log management
  - Service Registry
  - API Gateways
  - Extensive Monitoring
  - Service Routing
  - Flow control mechanisms



Service Routing / API Gateway

Service Logging & Monitoring

Microservices

Service Registry

DevOps

Self Managed, Self Healing Cloud Environment

# Antifragility, fail fast, and self-healing

- **Antifragility** is a technique successfully experimented at Netflix.
- It is one of the most powerful approaches to building fail-safe systems in modern software development.
- **Fail fast** is another concept used to build fault-tolerant, resilient systems
- **Self-healing** is commonly used in microservices deployments, where the system automatically learns from failures and adjusts itself. These systems also prevent future failures.

# Day - 1

# Building Microservices with Spring Boot

# Building microservices with spring boot

- There is no "one size fits all" approach when implementing microservices.
- Examining the simple microservices version of this application, we can immediately note a few things in this architecture:
  - Each subsystem has now become an independent system by itself, a microservice.
  - Each service encapsulates its own database as well as its own HTTP listener.
  - Each microservice exposes a REST service to manipulate the resources/entity that belong to this service.

# Microservices Benefits

- Supports Polygot Architecture
- Enabling experimentation and innovation
- Elastically and selectively scalable
- Allowing substitution
- Enabling to build organic systems
- Helping reducing technology debt
- Allowing the coexistence of different versions
- Supporting the building of self-organizing systems
- Supporting event-driven architecture
- Enabling DevOps

# Relationship with other architecture styles

- We will explore the relationship of microservices with other closely related architecture styles such as SOA and Twelve-Factor Apps

# Relations with SOA

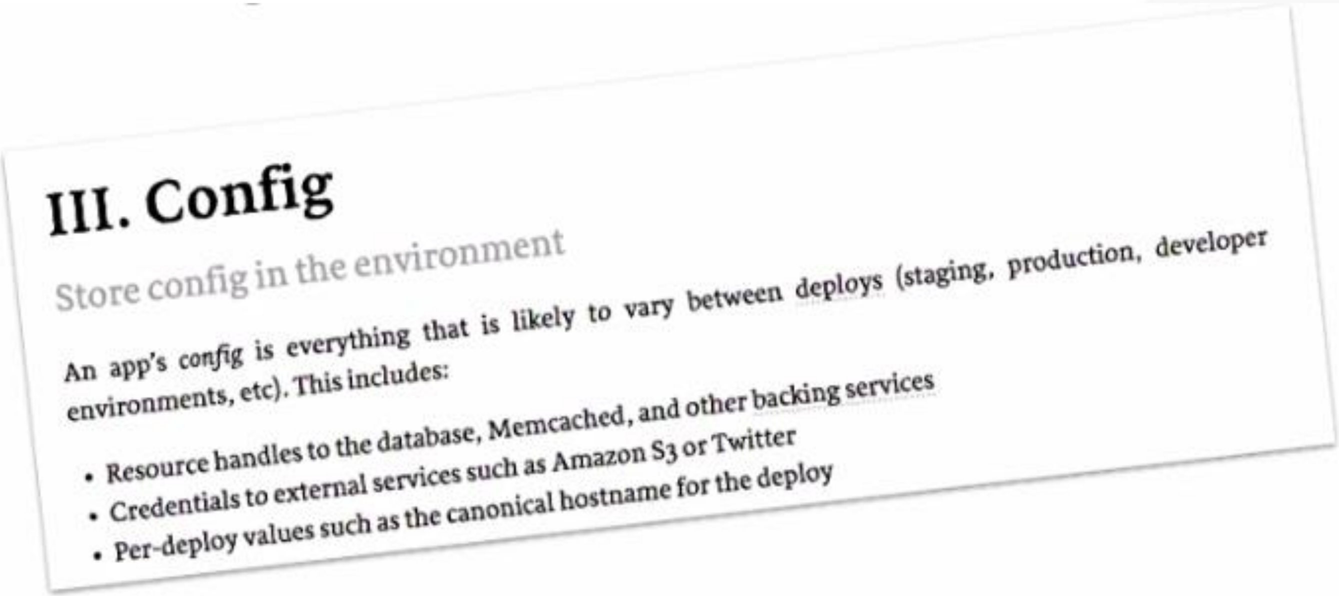
- The definition of SOA from The Open Group consortium is as follows:

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

- A service:
  - Is a logical representation of a repeatable business activity that has a specified outcome
  - It is self-contained.
  - It may be composed of other services.
  - It is a "black box" to consumers of the service."

# Relations with Twelve-Factor apps

- Twelve-Factor App, forwarded by Heroku, is a methodology describing the characteristics expected from modern cloud-ready applications.
- Twelve-Factor App is equally applicable for microservices as well. Hence, it is important to understand Twelve-Factor App.
- See 12factor.net/config



## III. Config

Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

# Microservices early adopters

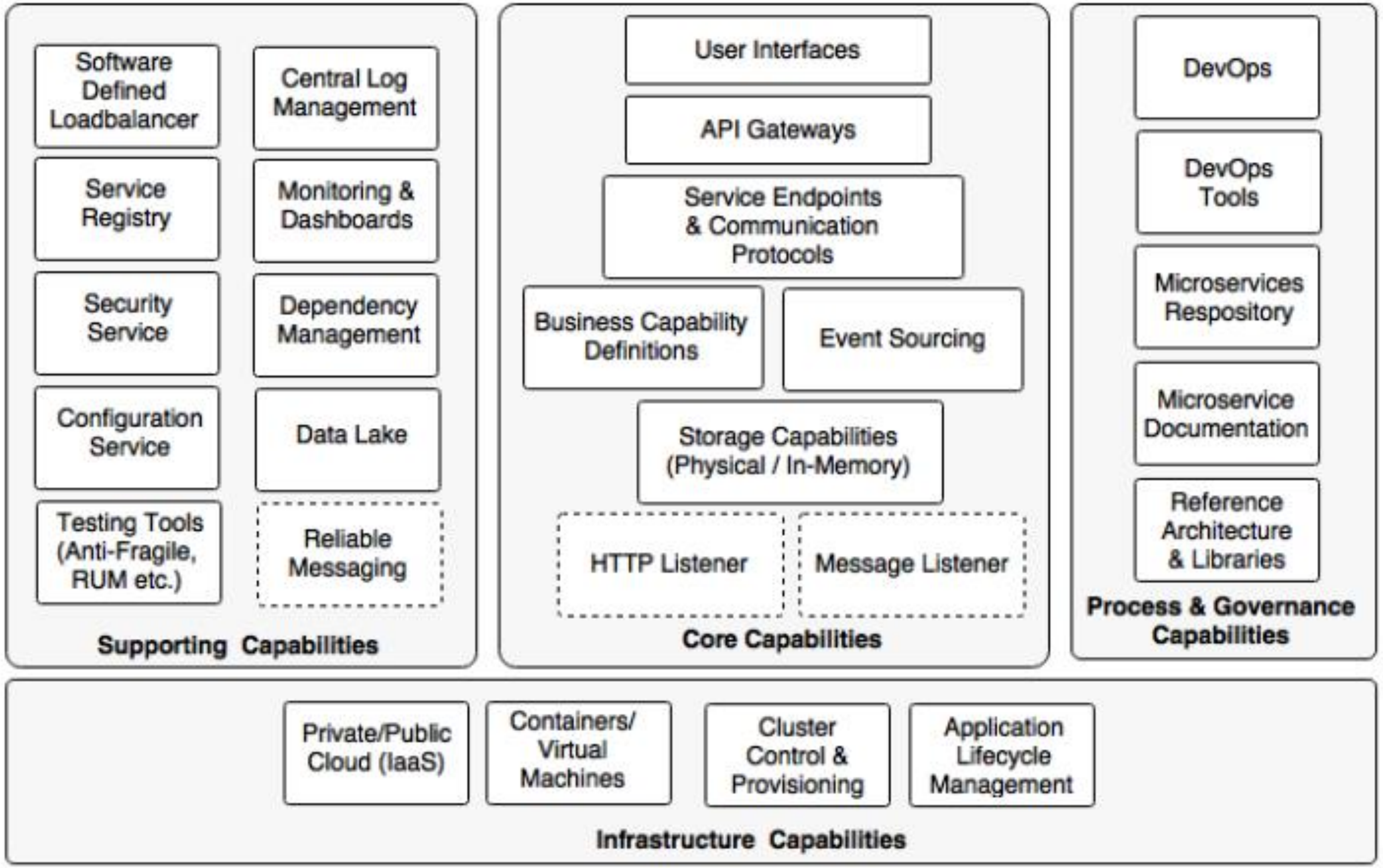
- Netflix
- Uber
- Airbnb
- Orbitz
- eBay
- Amazon
- Gilt
- Twitter
- Nike

Day - 1

# The Microservices Capability Model

# The microservices capability model

- The following diagram depicts the microservices capability model:



Software Defined Loadbalancer
Central Log Management
Service Registry
Monitoring & Dashboards
Security Service
Dependency Management
Configuration Service
Data Lake
Testing Tools (Anti-Fragile, RUM etc.)
Reliable Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process & Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Application Lifecycle Management
Infrastructure Capabilities

# The microservices capability model

The capability model is broadly classified in to four areas:

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

# Core capabilities

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

# Infrastructure capabilities

The Infrastructure capabilities are explained as follows:

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

# Supporting capabilities

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 CI management
- Reliable Messaging

# Process and governance capabilities

The Process and governance capabilities are explained as follows:

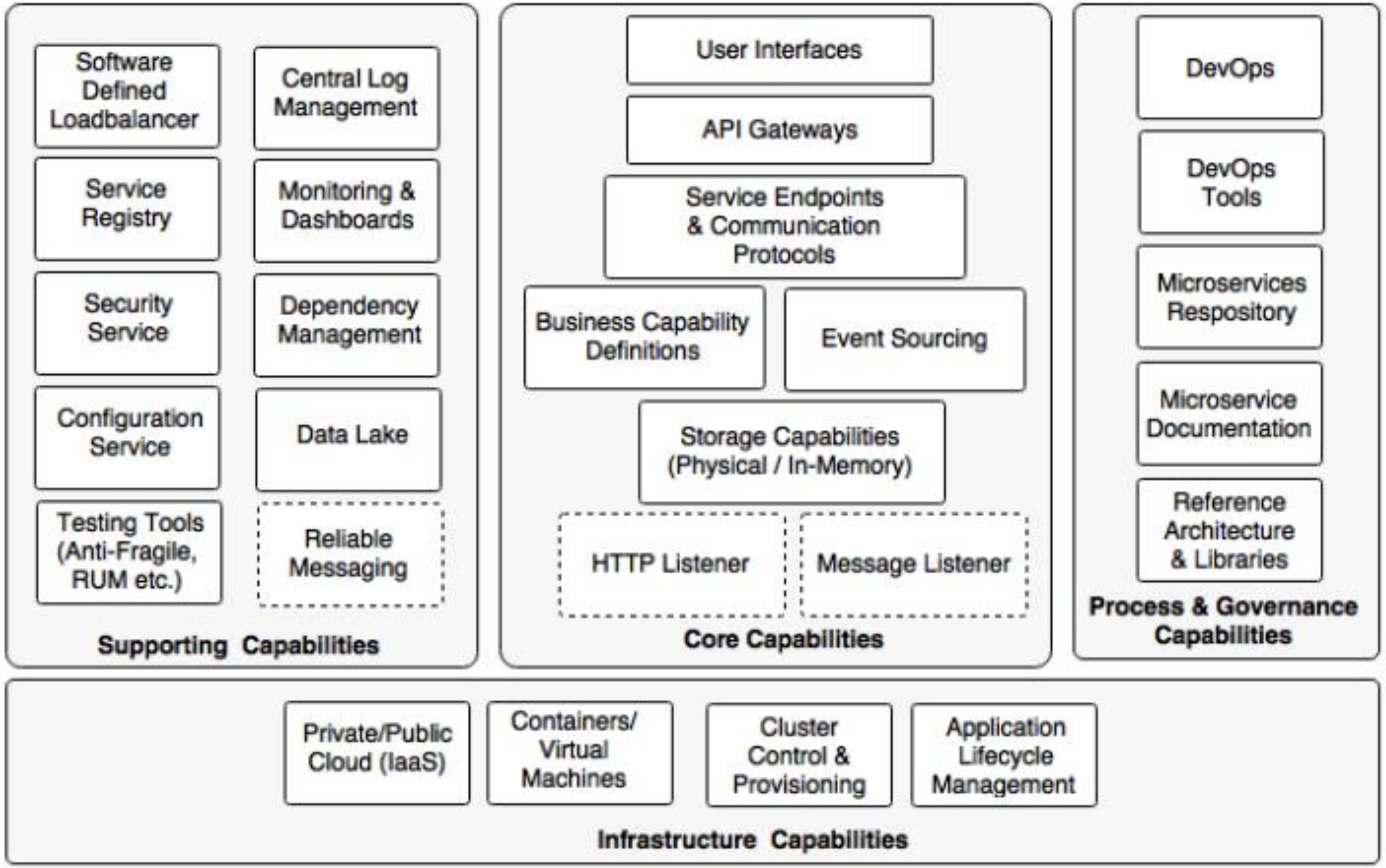
- DevOps
- DevOps tools
- Microservices repository
- Microservices documentation
- Reference architecture and libraries

# Reviewing microservices capabilities

We will explore the following microservices capabilities from the microservices capability model:

- Software Defined Load Balancer
- Service Registry
- Configuration Service
- Reliable Cloud Messaging
- API Gateways

# Reviewing microservices capabilities



Software Defined Loadbalancer
Central Log Management
Service Registry
Monitoring & Dashboards
Security Service
Dependency Management
Configuration Service
Data Lake
Testing Tools (Anti-Fragile, RUM etc.)
Reliable Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process & Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Application Lifecycle Management
Infrastructure Capabilities

# Recap of Day – 1

- What Is Monolithic Architecture?
- Concerns With the Monolith
- The Microservice architecture
- Characteristics of a Microservice Architecture
- Principles of microservices
- Business demand as a catalyst for Microservices
- Technology as a catalyst for the microservices evolution
- Building microservices with spring boot
- The microservices capability model

# Day - 2

# Day – 2 Agenda

- What is Spring Cloud?
- Components of Spring Cloud
- Spring Cloud Configuration – Centralized, Versioned Configuration
- Set up a Git repository
- Setting up the Config Server
- Accessing the Config Server from Clients
- Spring Cloud Config: How to use multiple configs

# Day - 2

# Spring Cloud

# What is Spring Cloud?

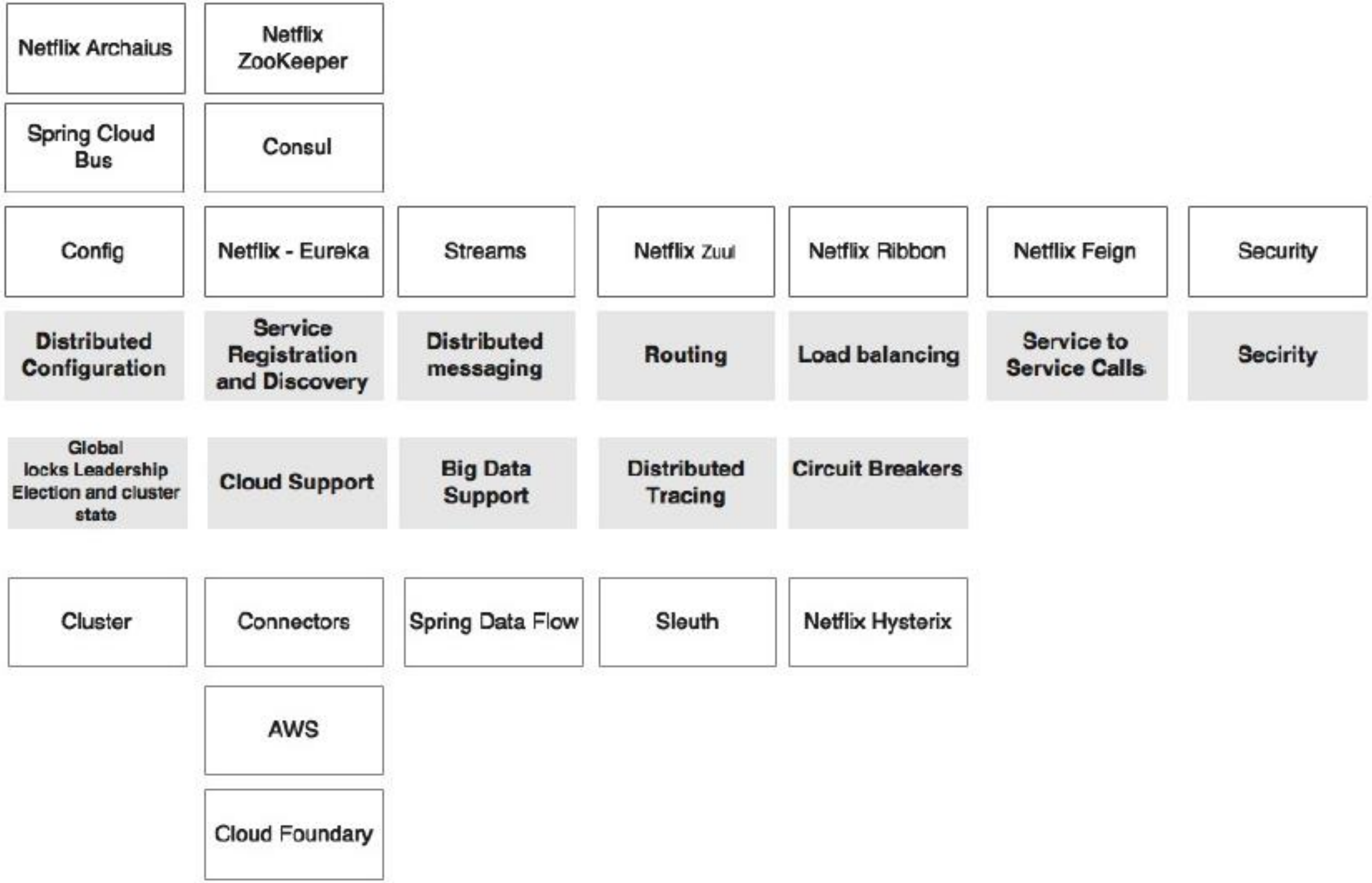
- Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g., configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).
- Coordination of distributed systems 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 centers, and managed platforms such as Cloud Foundry.

# Features

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others:

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging

# Components of Spring Cloud



| | | | | | | |
|---|---|---|---|---|---|---|
| Netflix Archaius | Netflix ZooKeeper | | | | | |
| Spring Cloud Bus | Consul | | | | | |
| Config | Netflix - Eureka | Streams | Netflix Zuul | Netflix Ribbon | Netflix Feign | Security |
| **Distributed Configuration** | **Service Registration and Discovery** | **Distributed messaging** | **Routing** | **Load balancing** | **Service to Service Calls** | **Secirity** |
| **Global locks Leadership Election and cluster state** | **Cloud Support** | **Big Data Support** | **Distributed Tracing** | **Circuit Breakers** | | |
| Cluster | Connectors | Spring Data Flow | Sleuth | Netflix Hysterix | | |
| | AWS | | | | | |
| | Cloud Foundary | | | | | |

# Spring Cloud and Netflix OSS

- Many of the Spring Cloud components which are critical for microservices deployment came from the **Netflix Open-Source Software (Netflix OSS)** center.
- Netflix is one of the pioneers and early adaptors in the microservices space. In order to manage large scale microservices, engineers at Netflix produced several homegrown tools and techniques for managing their microservices.
- Later, Netflix open-sourced these components, and made them available under the **Netflix OSS platform** for public use.
- These components are extensively used in production systems and are battle-tested with large scale microservice deployments at Netflix.
- Spring Cloud offers higher levels of abstraction for these Netflix OSS components, making it more Spring developer friendly. It also provides a declarative mechanism well-integrated and aligned with Spring Boot and the Spring framework.

# Day - 2

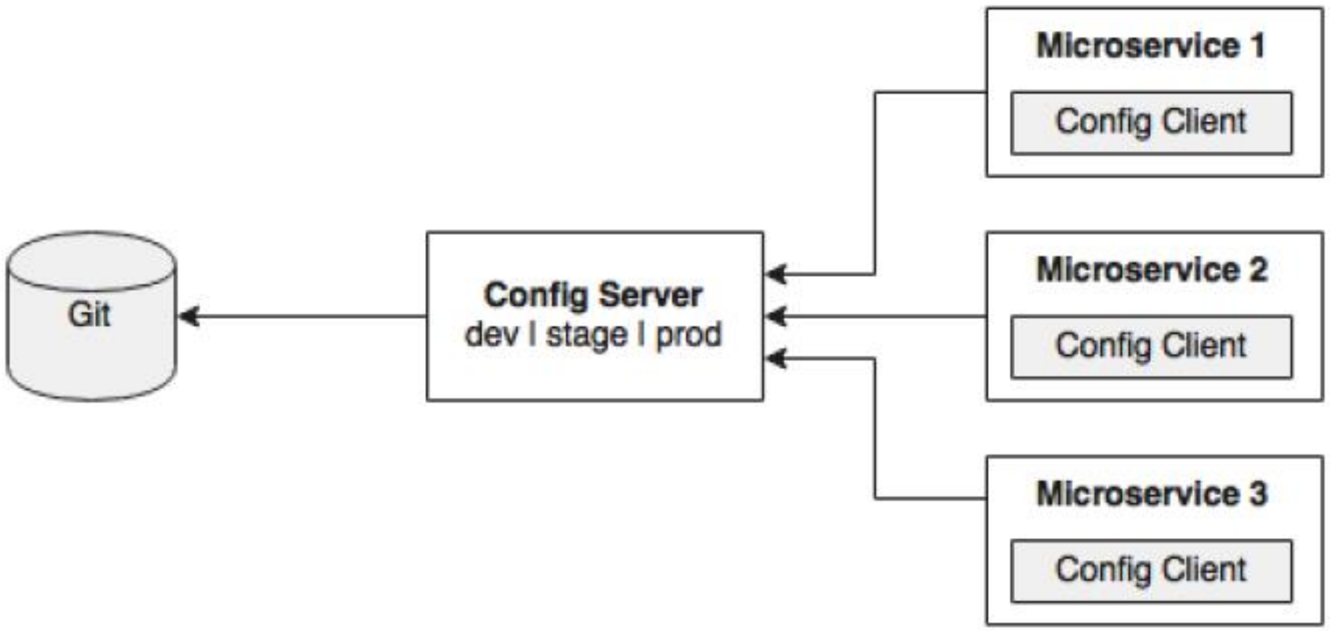
# Spring Cloud Config

# Spring Cloud Config

- The Spring Cloud Config server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties.
- The Spring Config server also supports version control of the configuration properties.

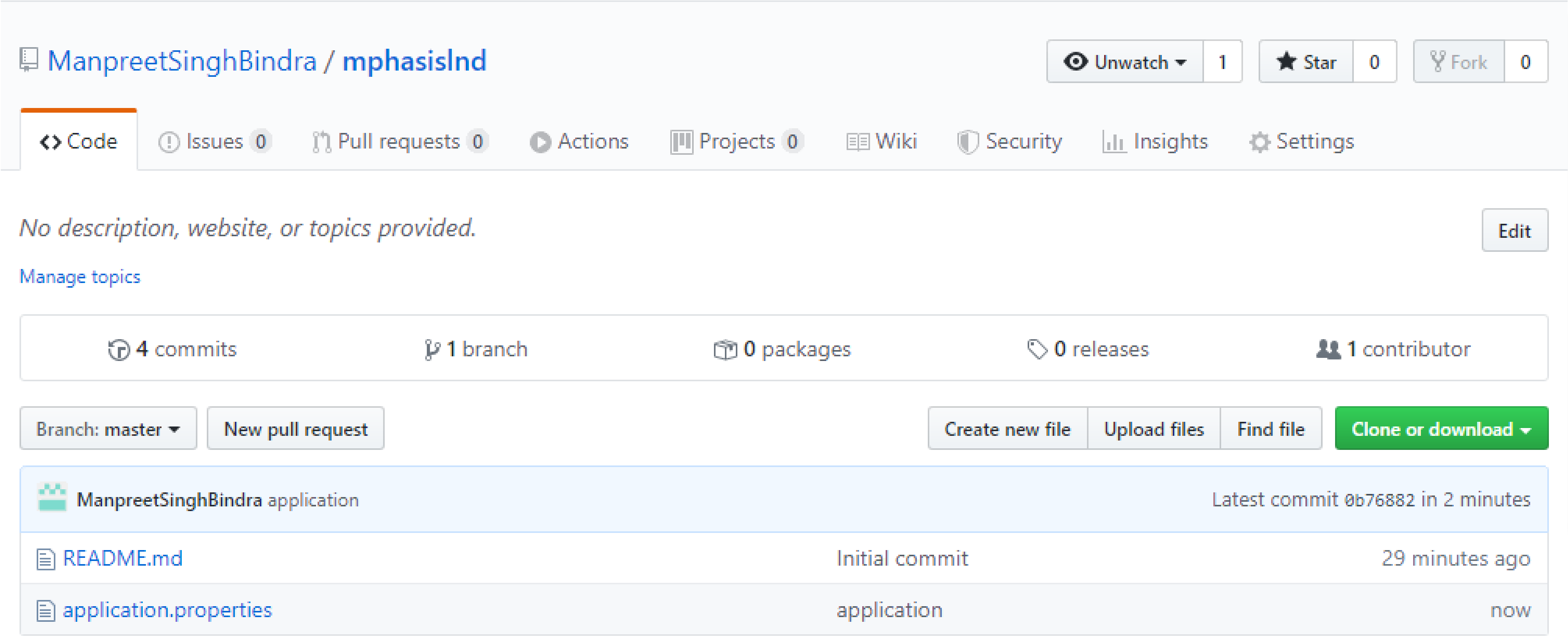
# Spring Cloud Config Server

- The Spring Config server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large scale distributed microservice deployments.
- The Spring Cloud Config server architecture is shown in the following diagram:



# Setting up the Config Server

1. Set up a Git repository and upload the application.properties file



ManpreetSinghBindra / mphasisInd

*No description, website, or topics provided.*

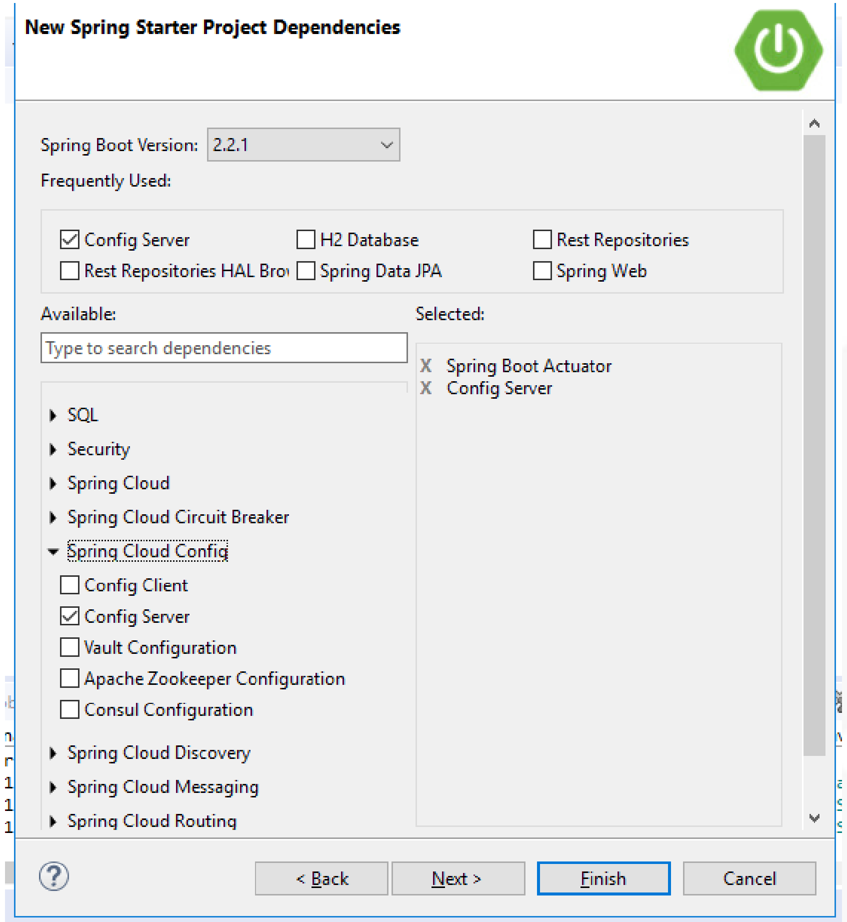
4 commits | 1 branch | 0 packages | 0 releases | 1 contributor

ManpreetSinghBindra application — Latest commit 0b76882 in 2 minutes

| | | |
|---|---|---|
| README.md | Initial commit | 29 minutes ago |
| application.properties | application | now |

# Setting up the Config Server

2. Create a new Spring Starter Project, and select Config Server and Actuator as shown in the following diagram:



New Spring Starter Project Dependencies

Spring Boot Version: 2.2.1

Frequently Used:

☑ Config Server □ H2 Database □ Rest Repositories
□ Rest Repositories HAL Bro□ □ Spring Data JPA □ Spring Web

Available:

Type to search dependencies

- SQL
- Security
- Spring Cloud
- Spring Cloud Circuit Breaker
- Spring Cloud Config
  - □ Config Client
  - ☑ Config Server
  - □ Vault Configuration
  - □ Apache Zookeeper Configuration
  - □ Consul Configuration
- Spring Cloud Discovery
- Spring Cloud Messaging
- Spring Cloud Routing

Selected:

X Spring Boot Actuator
X Config Server

# Setting up the Config Server

3. Add @EnableConfigServer in Application.java:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigserverApplication { … }
```

4. Edit the contents of the new application.properties file to match the following:

```
spring.port=8888
spring.cloud.config.server.git.uri=https://github.com/manpreetsinghbindra/
mphasisInd
```

# Setting up the Config Server

5. Run the Config server by right-clicking on the project, and running it as a Spring Boot app.

6. Check **http://localhost:8888/application/default/master** to see the properties specific to application.properties

# Accessing the Config Server from clients

- In the previous section, a Config server is set up and accessed using a web browser.
- In this section, the Product microservice will be modified to use the Config server. The Product microservice will act as a Config client.

# Accessing the Config Server from clients

1. Add the Spring Cloud Config and actuator dependency.

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

2. The new application.properties/bootstrap.properties file will look as follows:

```
server.port=8080
spring.cloud.config.uri=http://localhost:8888
```

3. Start the Config server.

4. Then start the Product microservice.

# Recap of Day – 2

- What is Spring Cloud?
- Components of Spring Cloud
- Spring Cloud Configuration – Centralized, Versioned Configuration
- Set up a Git repository
- Setting up the Config Server
- Accessing the Config Server from Clients
- Spring Cloud Config: How to use multiple configs

# Day - 3

# Day – 3 Agenda

- Spring Cloud Netflix
- Understanding Dynamic Service Registration and Discovery
- Understanding Eureka
- Setting up the Eureka server
- Enable dynamic registration and discovery to our microservice

# Day - 3

# Spring Cloud Netflix

# Spring Cloud Netflix

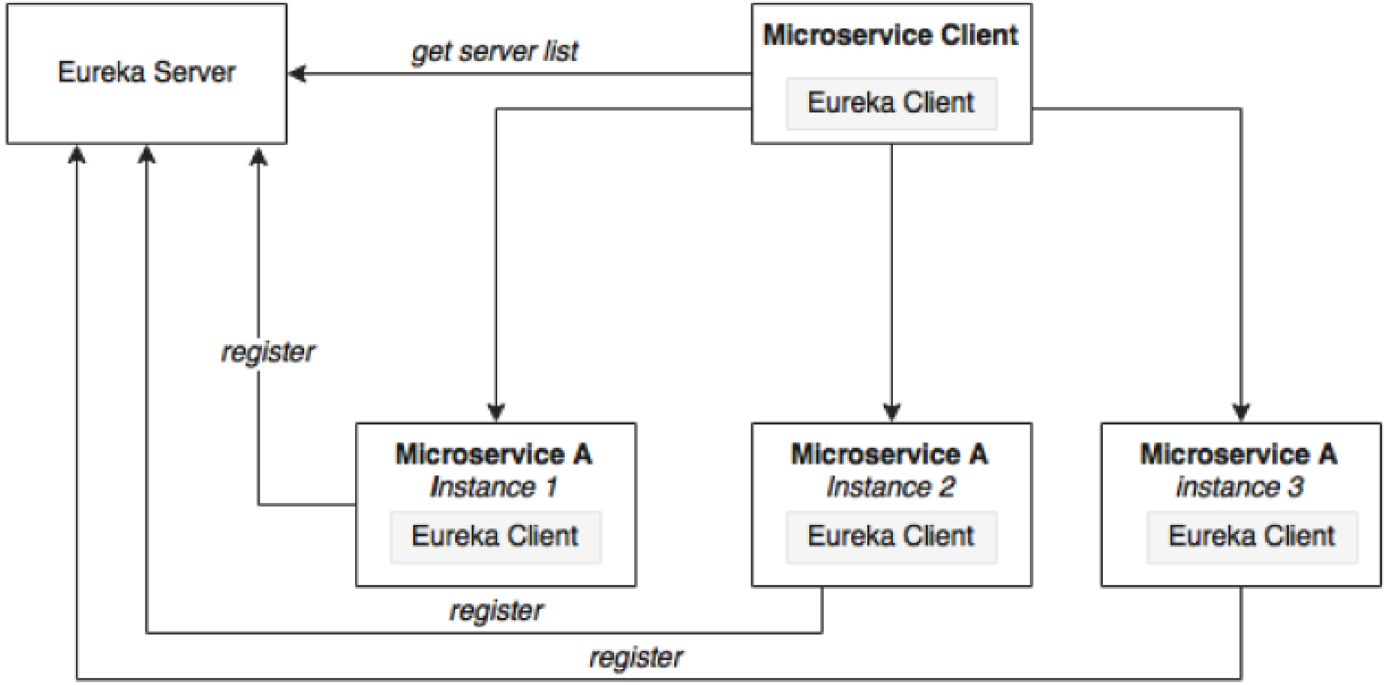
- Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.
- With a few simple annotations you can quickly enable and configure the common patterns inside your application and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

# 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.

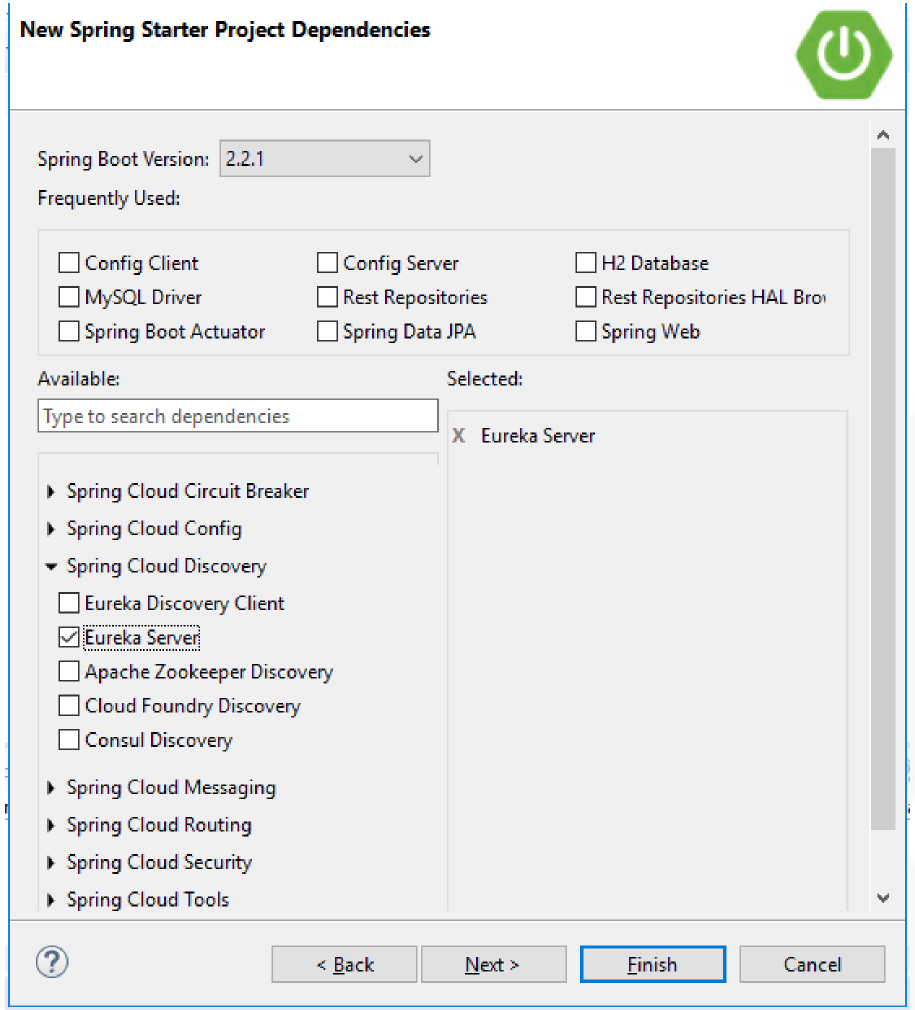
# Understanding Eureka

- 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 the Eureka server

1. Start a new Spring Starter project, and select Eureka Server, and Actuator:



**New Spring Starter Project Dependencies**

Spring Boot Version: 2.2.1

Frequently Used:

| | | |
|---|---|---|
| □ Config Client | □ Config Server | □ H2 Database |
| □ MySQL Driver | □ Rest Repositories | □ Rest Repositories HAL Brov |
| □ Spring Boot Actuator | □ Spring Data JPA | □ Spring Web |

Available:

Type to search dependencies

- Spring Cloud Circuit Breaker
- Spring Cloud Config
- Spring Cloud Discovery
  - □ Eureka Discovery Client
  - ☑ Eureka Server
  - □ Apache Zookeeper Discovery
  - □ Cloud Foundry Discovery
  - □ Consul Discovery
- Spring Cloud Messaging
- Spring Cloud Routing
- Spring Cloud Security
- Spring Cloud Tools

Selected:

X Eureka Server

# Setting up the Eureka server

2. Create a application.properties/bootstrap.properties

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

3. Add @EnableEurekaServer in Application.java:

```
@EnableEurekaServer

@SpringBootApplication

public class EurekaserverApplication { … }
```

4. We are now ready to start the Eureka server. Once the application is started, open http://localhost:8761 in a browser to see the Eureka console.

# Enable dynamic registration and discovery to our microservice

1. Add the Eureka dependencies to the pom.xml file

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

2. Create a application.properties/bootstrap.properties

```
spring.application.name=product-service
server.port=0
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://localhost:8761/eureka}
eureka.instance.preferIpAddress=true
```

# Enable dynamic registration and discovery to our microservice

3. Add @EnableEurekaClient in Application.java:

```
@EnableEurekaClient
@SpringBootApplication
public class EurekaclientApplication { … }
```

4. Start the Product Service.

# Recap of Day – 3

- Spring Cloud Netflix
- Understanding Dynamic Service Registration and Discovery
- Understanding Eureka
- Setting up the Eureka server
- Enable dynamic registration and discovery to our microservice

# Day - 4

# Day – 4 Agenda

- Client-Side Load Balancer: Ribbon
- Access from a client service

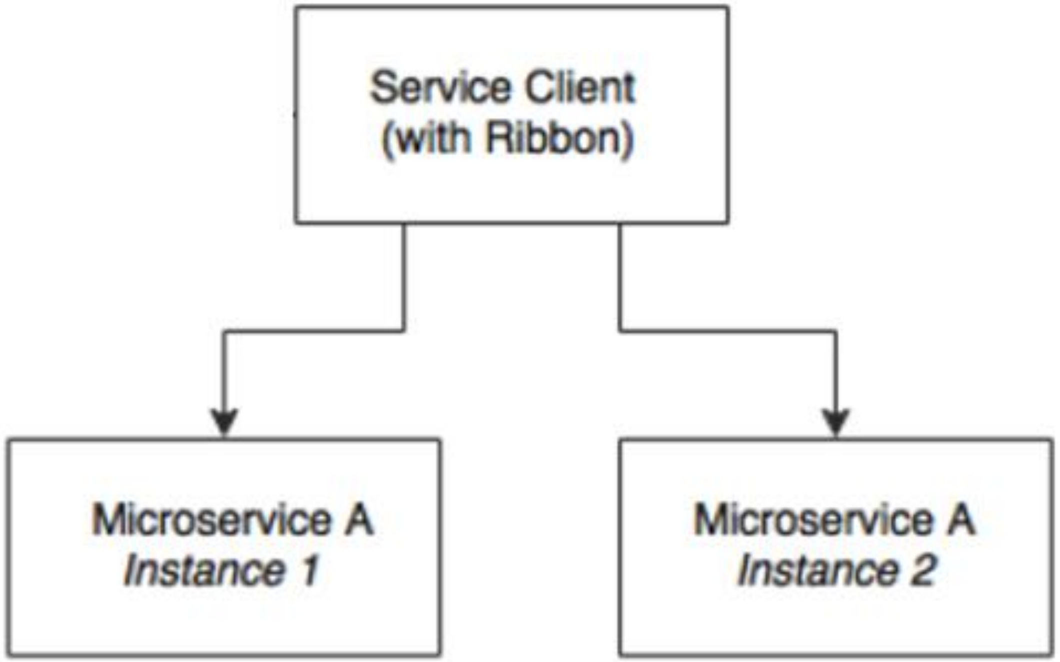
- Feign as a declarative REST client
- Create a ProductServiceProxy interface
- Have Service-to-Service calls
- Feign clients to be scanned and discovered

# Day - 4

# Client-Side Load Balancer: Ribbon

# Ribbon for load balancing

- Netflix Ribbon is an Inter Process Communication (IPC) cloud library.
- Ribbon primarily provides client-side load balancing algorithms.



# Ribbon for load balancing

- Ribbon provides also other features:
  - Service Discovery Integration
  - Fault Tolerance
  - Configurable load-balancing rules

# Ribbon for load balancing

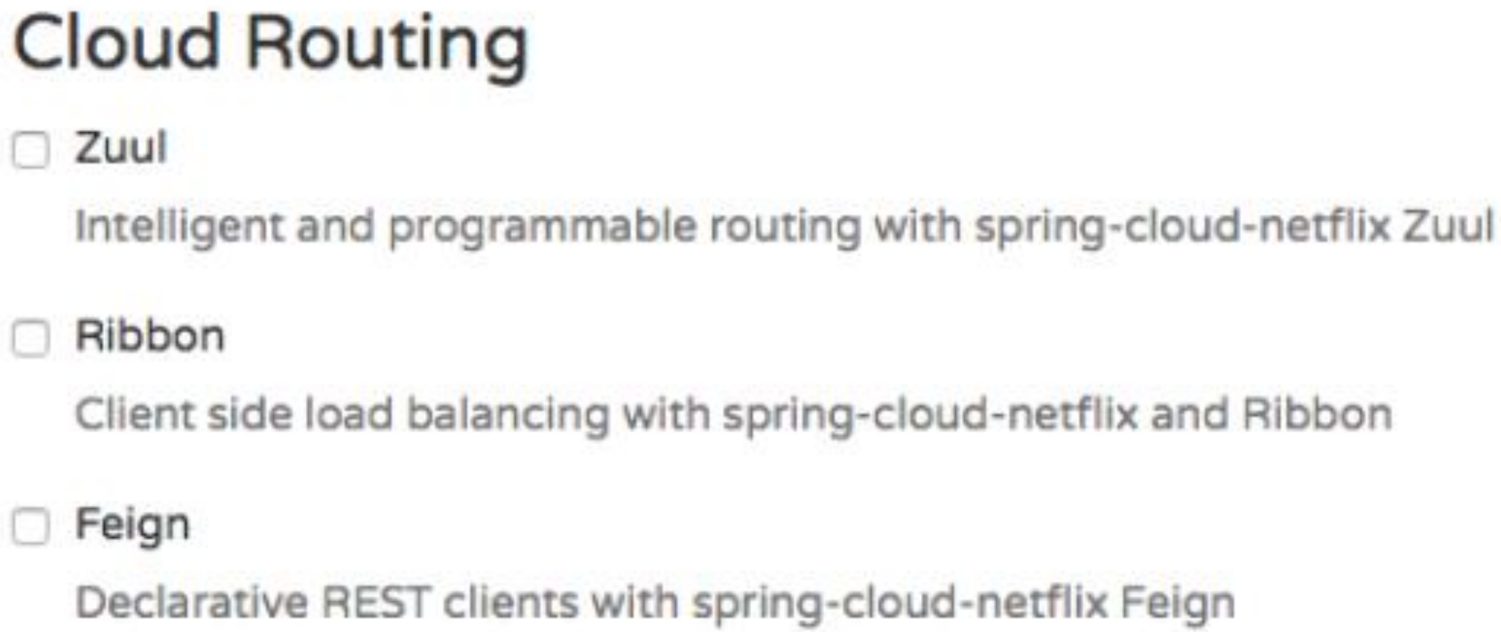
- In order to use the Ribbon client, we will have to add the following dependency to the pom.xml file:

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- In case of development from ground up, this can be selected from the Spring Starter libraries, or from http://start.spring.io/.

# Ribbon for load balancing

- Ribbon is available under **Cloud Routing**:



## Cloud Routing

□ **Zuul**

Intelligent and programmable routing with spring-cloud-netflix Zuul

□ **Ribbon**

Client side load balancing with spring-cloud-netflix and Ribbon

□ **Feign**

Declarative REST clients with spring-cloud-netflix Feign

# Client-side vs server-side load balancing

- The multiple instances of the same microservice is 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.

# Spring client-side load balancing

- Spring RestTemplate can be used for client-side load balancing
- Spring Netflix Eureka has a built-in client-side load balancer called Ribbon.
- Ribbon can automatically be configured by registering RestTemplate as a bean and annotating it with @LoadBalanced.

```
@Configuration
public class Config {
        @LoadBalanced
        @Bean
        public RestTemplate restTemplate() {
                return new RestTemplate();
        }       }
```

# Access from a client service

```
@RestController
@Scope("request")
public class ProductClientController {
        @Autowired
        private RestTemplate restTemplate;
        @GetMapping("/get-products/{id}")
        public Product getProductById(@PathVariable("id") int id) {
                Product product = restTemplate.getForObject("http://product-service/products/"+id, Product.class);
                return product;
        }
}
```

# Day - 4

# Spring Cloud OpenFeign

# Feign as a declarative REST client

- Feign is a Spring Cloud Netflix library for providing a higher level of abstraction over REST-based service calls.
- Spring Cloud Feign offers a declarative approach for making RESTful service-to-service call in a synchronous way.
- When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client.
- The developer need not worry about the implementation of this interface.
- This will be dynamically provisioned by Spring at runtime.
- With this declarative approach, developers need not get into the details of the HTTP level APIs provided by RestTemplate.

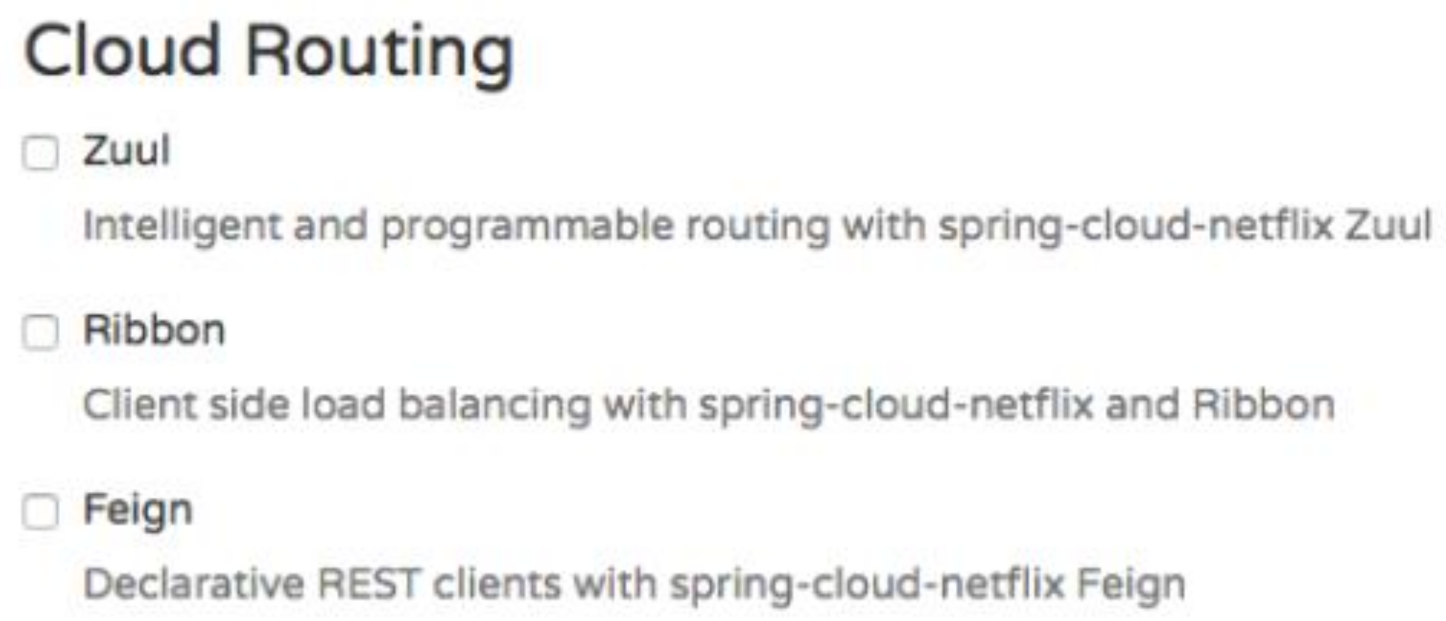
# Feign as a declarative REST client

- In order to use Feign, first we need to change the pom.xml file to include the Feign dependency as follows:

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

# Feign as a declarative REST client

- For a new Spring Starter project, Feign can be selected from the starter library selection screen, or from [http://start.spring.io/](http://start.spring.io/).
- This is available under **Cloud Routing** as shown in the following screenshot:



Cloud Routing

□ Zuul

Intelligent and programmable routing with spring-cloud-netflix Zuul

□ Ribbon

Client side load balancing with spring-cloud-netflix and Ribbon

□ Feign

Declarative REST clients with spring-cloud-netflix Feign

# Create a ProductServiceProxy interface

- The next step is to create a new ProductServiceProxy interface.
- This will act as a proxy interface of the actual Product service:

```
@FeignClient("product-service")
public interface ProductServiceProxy {

    @GetMapping(value = "/products/{id}", produces = {
                MediaType.APPLICATION_JSON_VALUE})
    public Product getProductById(@PathVariable("id") int id);

    @GetMapping(value = "/products", produces = {
                MediaType.APPLICATION_JSON_VALUE})
    public ArrayList<Product> getAllProducts();
}
```

# Have Service-to-Service calls

- The next step is to create a new ProductClientController class

```
@RestController
@Scope("request")
public class ProductClientController {
    @Autowired
    private ProductServiceProxy productServiceProxy;

    @GetMapping("/get-products/{id}")
    public Product getProductById(@PathVariable("id") int id) {

                Product product = productServiceProxy.getProductById(id);
                return product;
    }
}
```

# Feign clients to be scanned and discovered

- Add @EnableFeignClients in Application.java:

```
@EnableFeignClients
@SpringBootApplication
public class FeignclientApplication { … }
```

# Recap of Day – 4

- Client-Side Load Balancer: Ribbon
- Access from a client service

- Feign as a declarative REST client
- Create a ProductServiceProxy interface
- Have Service-to-Service calls
- Feign clients to be scanned and discovered

# Day - 5

# Day – 5 Agenda

- Isolating from failures
- Spring Cloud Hystrix for fault-tolerant microservices

# Day - 5

# Isolating from failures

# Circuit breaker

- The circuit breaker subproject implements the circuit breaker pattern.
- The circuit breaker breaks the circuit when it encounters failures in the primary service by diverting traffic to another temporary fallback service.
- It also automatically reconnects back to the primary service when the service is back to normal.
- It finally provides a monitoring dashboard for monitoring the service state changes.
- The Spring Cloud Hystrix project and Hystrix Dashboard implement the circuit breaker and the dashboard, respectively.

# Spring Cloud Hystrix for fault-tolerant microservices

- Spring Cloud Hystrix as a library for a fault-tolerant and latency-tolerant microservice implementation.
- Hystrix is based on the fail-fast and rapid recovery principles. If there is an issue with a service, Hystrix helps isolate it. It helps to recover quickly by falling back to another preconfigured fallback service.
- Hystrix is another battle-tested library from Netflix. Hystrix is based on the circuit breaker pattern.

# Build a circuit breaker with Spring Cloud Hystrix

1. Add the Hystrix dependency to the service

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

2. In the Spring Boot Application class, add **@EnableCircuitBreaker.** This command will tell Spring Cloud Hystrix to enable a circuit breaker for this application. It also exposes the /hystrix.stream endpoint for metrics collection.

# @HystrixCommand

- @HystrixCommand tells Spring that this method is prone to failure
- Spring Cloud libraries wrap these methods to handle fault tolerance and latency tolerance by enabling circuit breaker.
- The Hystrix command typically follows with a **fallback method**.
- In case of failure, Hystrix automatically enables the fallback method mentioned and diverts traffic to the fallback method.

# @HystrixCommand

```
@Service
public class ProductClientService {
    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "getDefaultProductById")
    public Product getProductById(int id) {
                Product product = restTemplate.getForObject("http://product-
service/products/" + id, Product.class);
                return product;
    }
    public Product getDefaultProductById(int id) {
                return new Product(id, "Sony", 88888.0);
    }
}
```

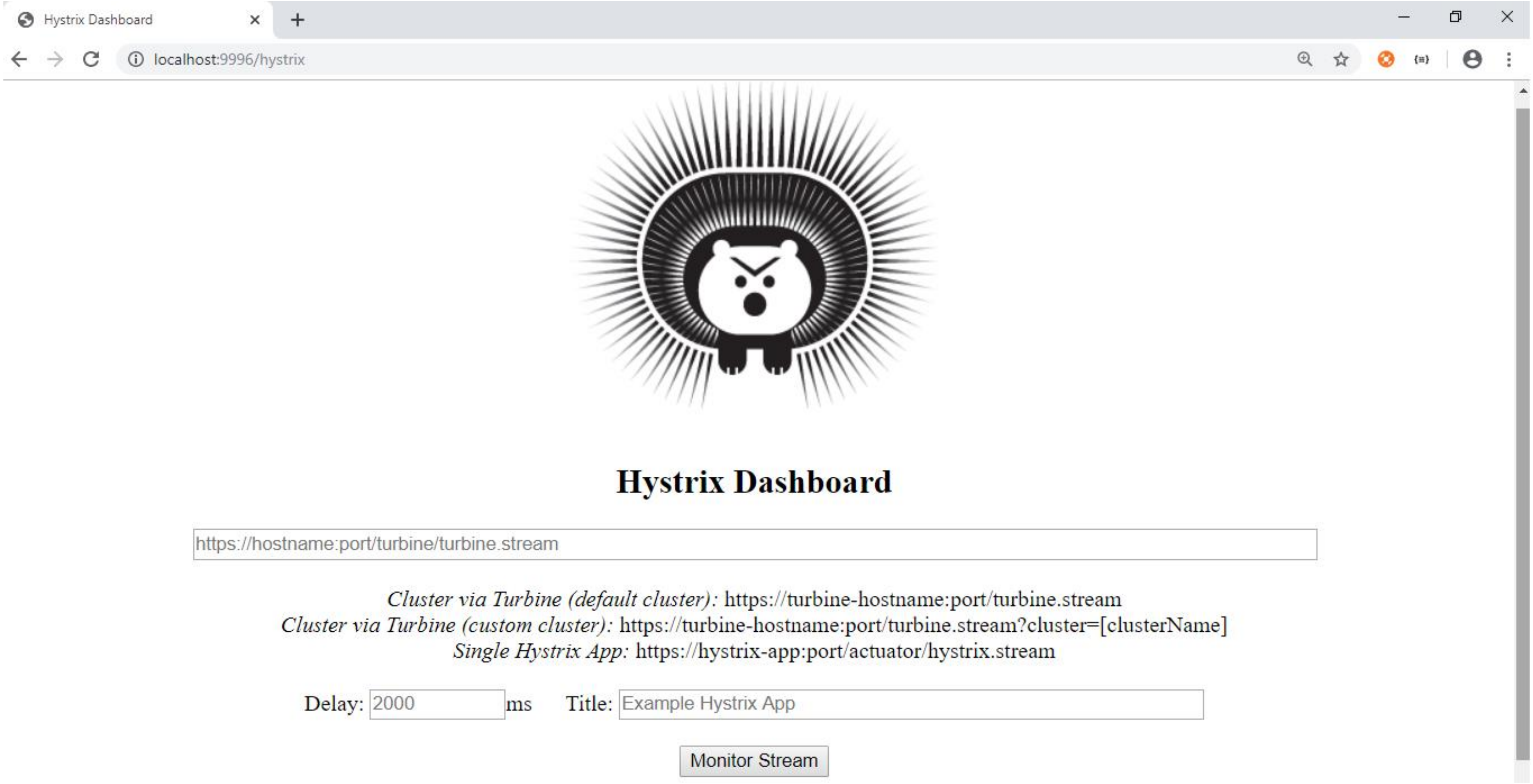
# Build a Hystrix Dashboard

1. Add the Hystrix, Hystrix Dashboard, and Actuator dependency to the application

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

2. In the Spring Boot Application class, add the **@EnableHystrixDashboard** annotation.
3. Start the Product service, Search API Gateway and Hystrix Dashboard application.
4. The Hystrix Dashboard is started on application port.

# Hystrix Dashboard



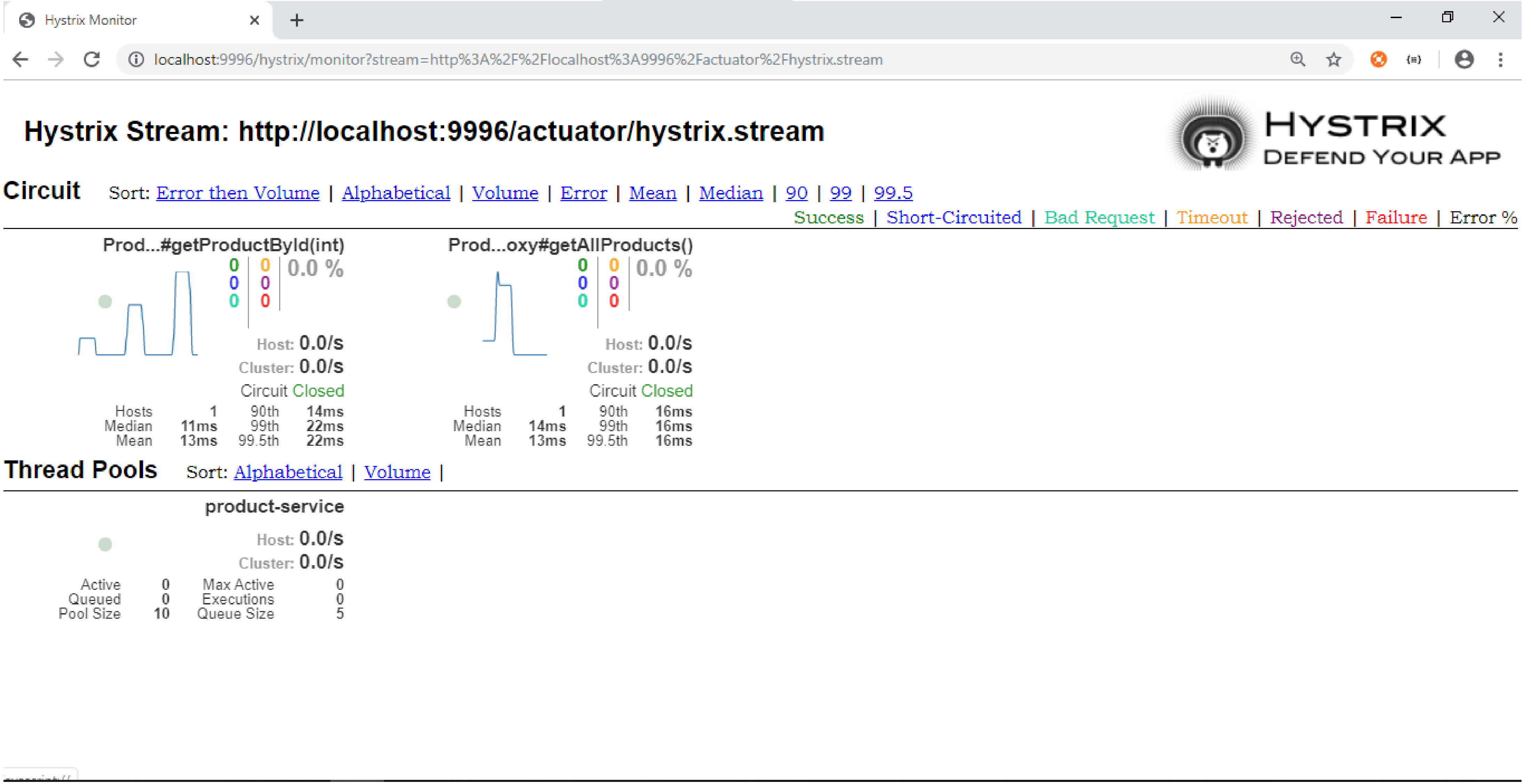
Hystrix Dashboard

https://hostname:port/turbine/turbine.stream

*Cluster via Turbine (default cluster):* https://turbine-hostname:port/turbine.stream
*Cluster via Turbine (custom cluster):* https://turbine-hostname:port/turbine.stream?cluster=[clusterName]
*Single Hystrix App:* https://hystrix-app:port/actuator/hystrix.stream

Delay: 2000 ms Title: Example Hystrix App

# Hystrix Dashboard



Hystrix Monitor
localhost:9996/hystrix/monitor?stream=http%3A%2F%2Flocalhost%3A9996%2Factuator%2Fhystrix.stream
HYSTRIX
DEFEND YOUR APP

## Hystrix Stream: http://localhost:9996/actuator/hystrix.stream

**Circuit** Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

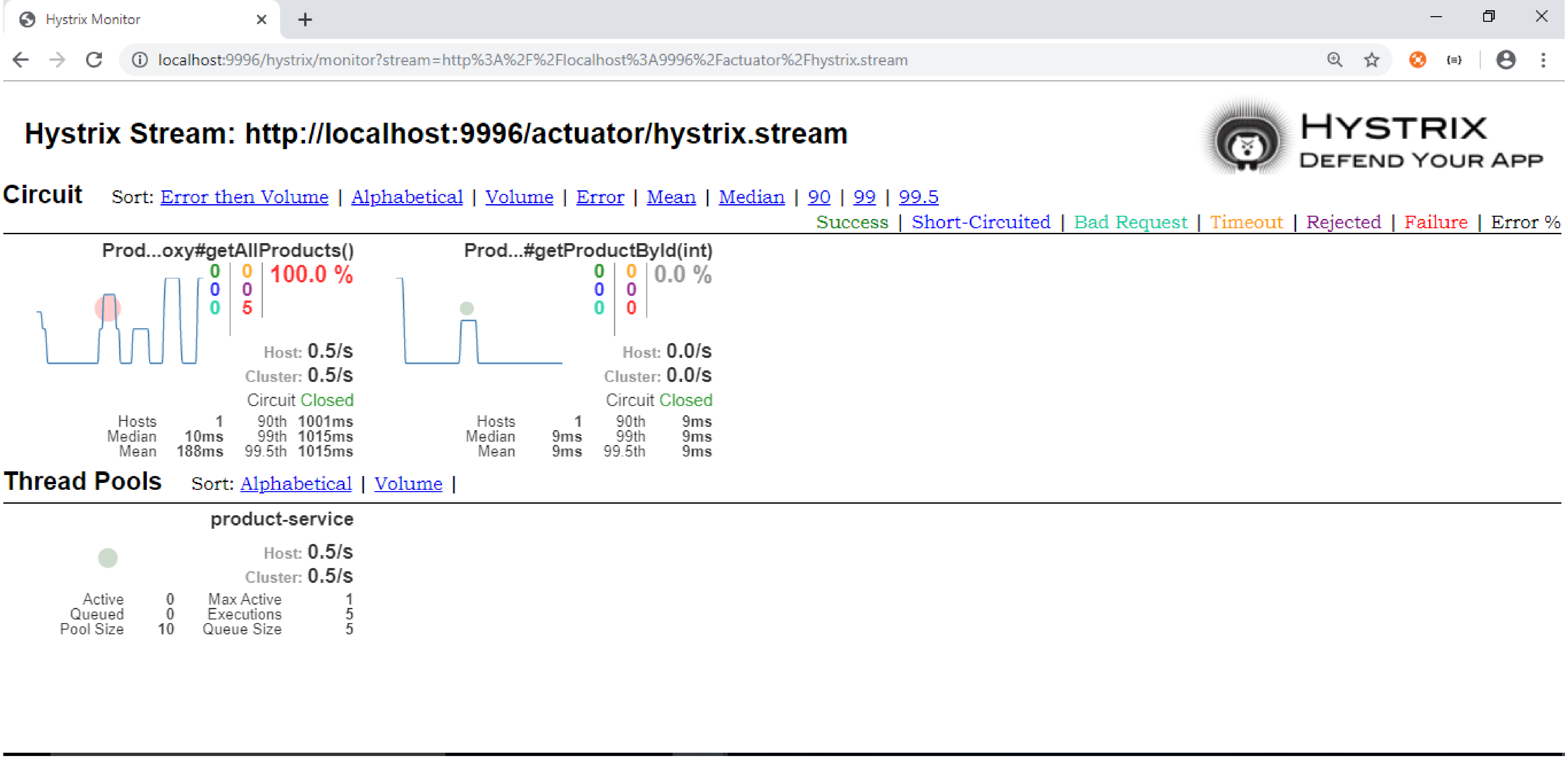
Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Error %

```
Prod...#getProductById(int)                 Prod...oxy#getAllProducts()
        0 | 0 | 0.0 %                               0 | 0 | 0.0 %
        0 | 0                                       0 | 0
        0 | 0                                       0 | 0
      Host: 0.0/s                                 Host: 0.0/s
   Cluster: 0.0/s                              Cluster: 0.0/s
    Circuit Closed                              Circuit Closed
 Hosts      1    90th    14ms              Hosts      1    90th    16ms
 Median  11ms    99th    22ms              Median  14ms    99th    16ms
 Mean    13ms  99.5th    22ms              Mean    13ms  99.5th    16ms
```

**Thread Pools** Sort: Alphabetical | Volume |

```
                          product-service
                      Host: 0.0/s
                   Cluster: 0.0/s
   Active      0   Max Active    0
   Queued      0   Executions    0
   Pool Size  10   Queue Size    5
```

# Hystrix Dashboard



localhost:9996/hystrix/monitor?stream=http%3A%2F%2Flocalhost%3A9996%2Factuator%2Fhystrix.stream

**Hystrix Stream: http://localhost:9996/actuator/hystrix.stream**

**Circuit** Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Error %

```
Prod...oxy#getAllProducts()            Prod...#getProductById(int)
 0   0  100.0 %                          0   0  0.0 %
 0   0                                   0   0
 0   5                                   0   0
          Host: 0.5/s                             Host: 0.0/s
       Cluster: 0.5/s                          Cluster: 0.0/s
        Circuit Closed                          Circuit Closed
 Hosts       1    90th  1001ms          Hosts     1    90th   9ms
 Median   10ms    99th  1015ms          Median  9ms    99th   9ms
 Mean    188ms  99.5th  1015ms          Mean    9ms  99.5th   9ms
```

**Thread Pools** Sort: Alphabetical | Volume |

```
                  product-service
               Host: 0.5/s
            Cluster: 0.5/s
 Active       0   Max Active    1
 Queued       0   Executions    5
 Pool Size   10   Queue Size    5
```

# Recap of Day – 5

- Isolating from failures
- Spring Cloud Hystrix for fault-tolerant microservices

# Day - 6

# Day – 6 Agenda

- What is API Gateway?
- Spring cloud routing – Zuul
- Setting up Zuul

- Autoscaling microservices
- Scaling microservices with Spring Cloud
- Understanding the concept of autoscaling
- The benefits of autoscaling
- Different autoscaling models
- Autoscaling Approaches

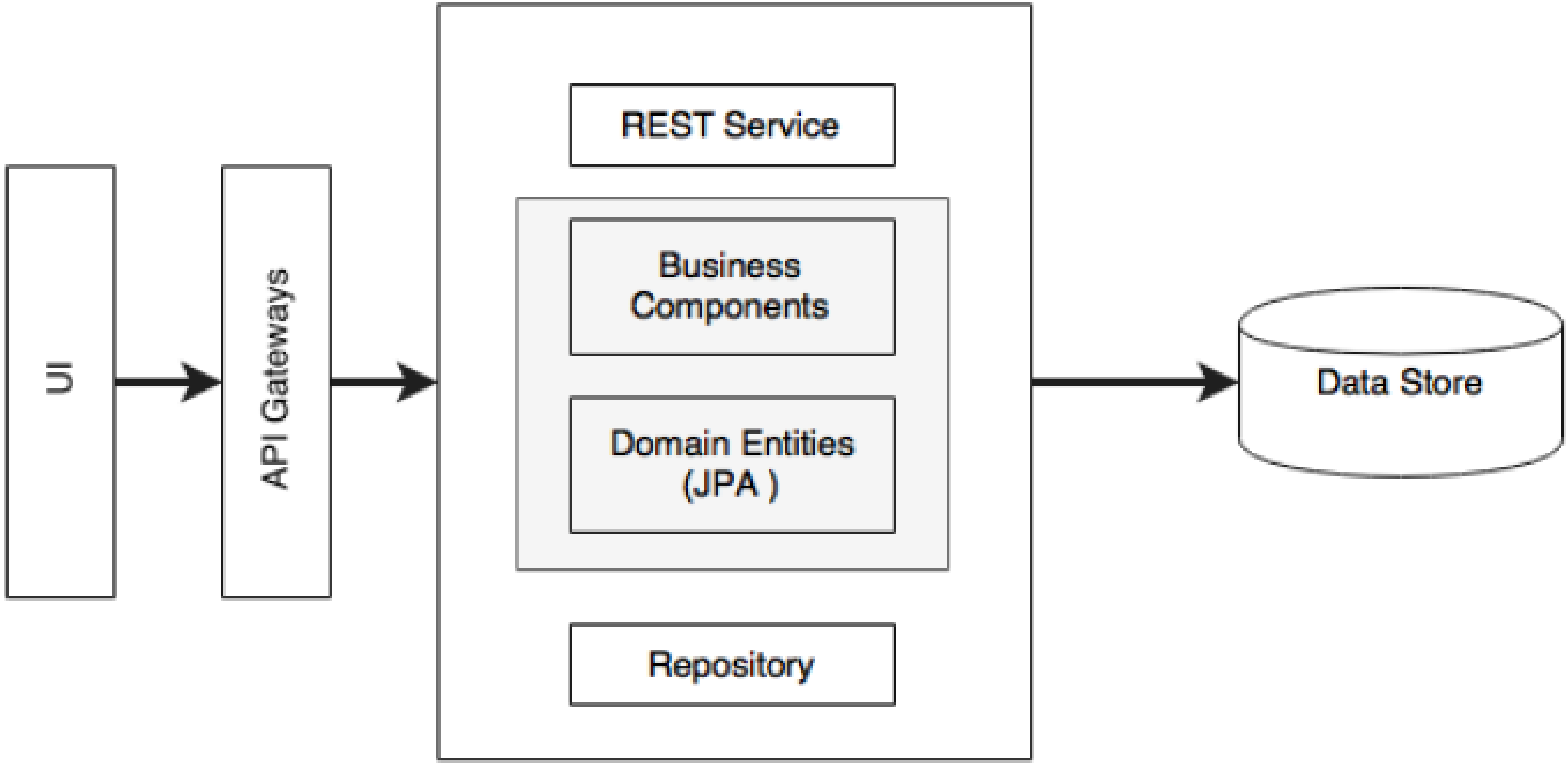
# Day - 6

# Spring cloud routing - Zuul

# API Gateway

- An **API Gateways**, aka **Edge Service**, provides a unified interface for a set of microservices so that client no need to know about all the details of microservices internals.
- The API gateway provides a level of indirection by either proxying service endpoints or composing multiple service endpoints.
- The API gateway is also useful for policy enforcements.
- It may also provide real time load balancing capabilities.
- There are many API gateways available in the market.
- Spring Cloud Zuul, Mashery, Apigee, and 3scale are some examples of the API gateway providers.

# API Gateway

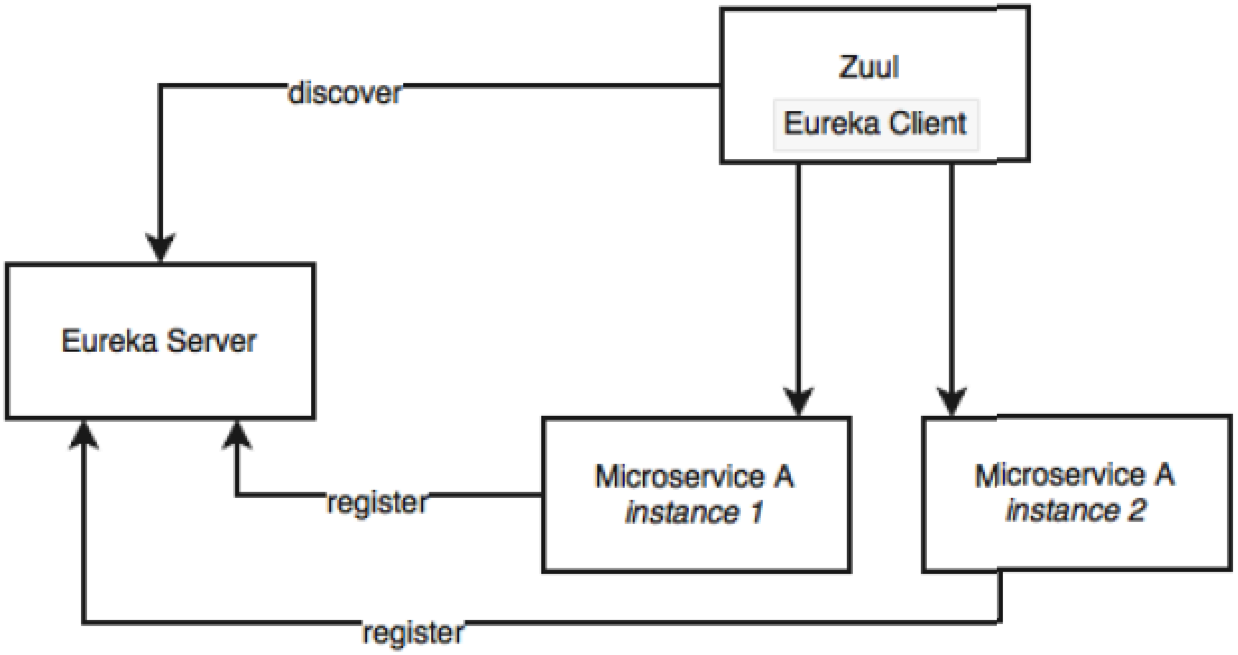

UI
API Gateways
REST Service
Business Components
Domain Entities (JPA )
Repository
Data Store

# Spring Cloud Routing

- Routing is an API gateway component, primarily used like a reverse proxy that forwards requests from consumers to service providers.
- The gateway component can also perform software-based routing and filtering.
- Zuul is a lightweight API gateway solution that offers fine-grained controls to developers for traffic shaping and request/response transformations.

# Zuul proxy as the API Gateway

- Zuul is a simple gateway service or edge service that suits these situations well.
- Zuul also comes from the Netflix family of microservice products.
- Unlike many enterprise API gateway products, Zuul provides complete control for the developers to configure, or program based on specific requirements:



# Zuul proxy as the API Gateway

- The Zuul proxy internally uses the Eureka server for service discovery, and Ribbon for load balancing between service instances.
- The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on.
- In simple terms, we can consider Zuul a reverse proxy service. With Zuul, we can even change the behaviors of the underlying services by overriding them at the API layer.

# Zuul Components

- Zuul has mainly four types of filters that enable us to intercept the traffic in different timeline of the request processing for any particular transaction.
- We can add any number of filters for a url pattern.
  - **pre filters** – are invoked before the request is routed.
  - **post filters** – are invoked after the request has been routed.
  - **route filters** – are used to route the request.
  - **error filters** – are invoked when an error occurs while handling the request.

# Setting up Zuul

1. Add the Zuul dependency to the application

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

2. In the Spring Boot Application class, add the @EnableZuulProxy annotation.
3. Create the different types of custom filters extending ZuulFilter:
   - PreFilter
   - PostFilter
   - RouteFilter
   - ErrorFilter

# Setting up Zuul

4. This configuration also sets a rule on how to forward traffic. In this case, any request coming on the /api endpoint of the API gateway should be sent to product-service:

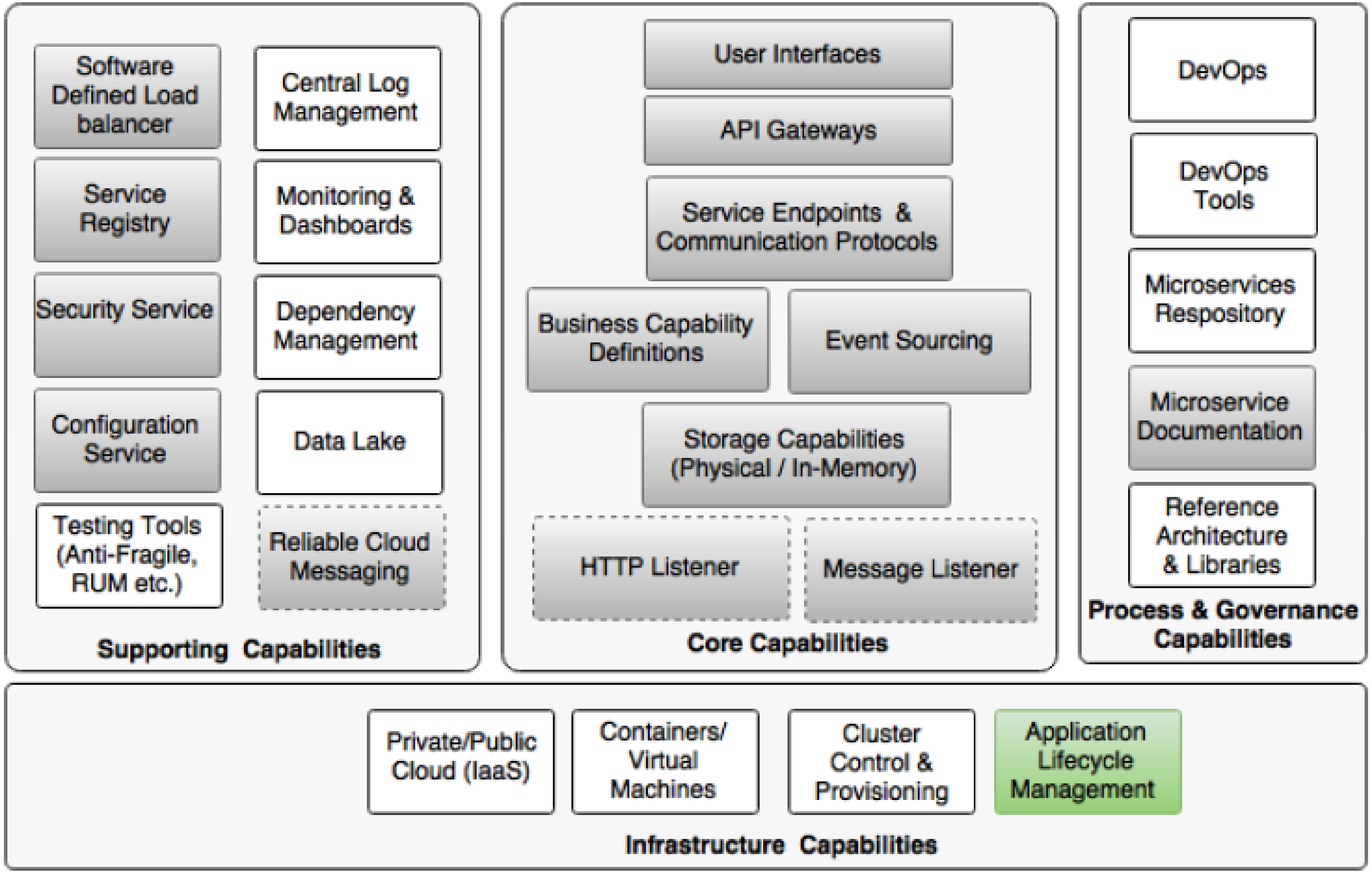
```
#Zuul routes.
zuul.routes.products.url=http://localhost:56024/
zuul.routes.products.path=/api/demo/**

#Will start the gateway server @8080
server.port=8080
```

# Day - 6

# Autoscaling Microservices

# Reviewing the microservice capability model



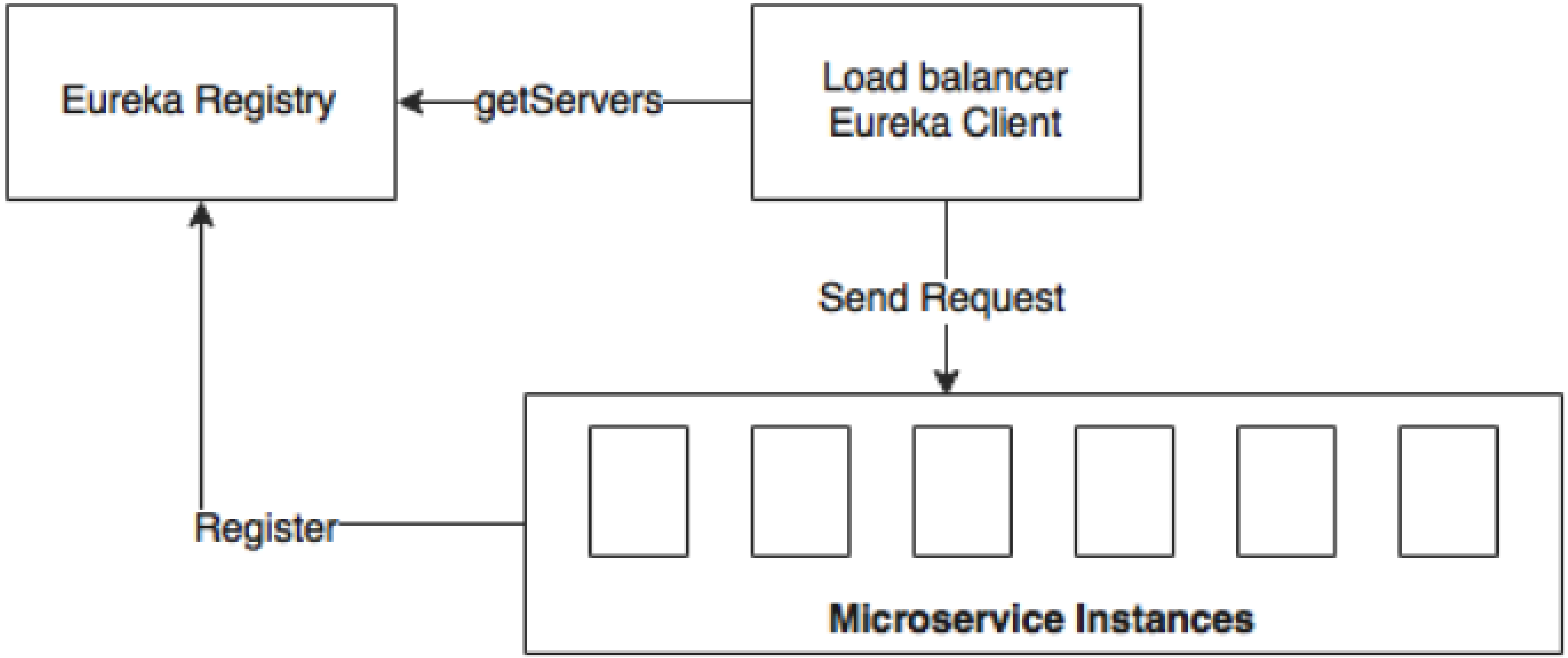
Software Defined Load balancer
Central Log Management
Service Registry
Monitoring & Dashboards
Security Service
Dependency Management
Configuration Service
Data Lake
Testing Tools (Anti-Fragile, RUM etc.)
Reliable Cloud Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process & Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Application Lifecycle Management
Infrastructure Capabilities

# Scaling microservices with Spring Cloud

- The two key concepts of Spring Cloud that we implemented are **self-registration** and **self-discovery.**
- These two capabilities enable automated microservices deployments.
- With self-registration, microservices can automatically advertise the service availability by registering service metadata to a central service registry as soon as the instances are ready to accept traffic.
- Once the microservices are registered, consumers can consume the newly registered services from the very next moment by discovering service instances using the registry service. Registry is at the heart of this automation.

# Scaling microservices with Spring Cloud

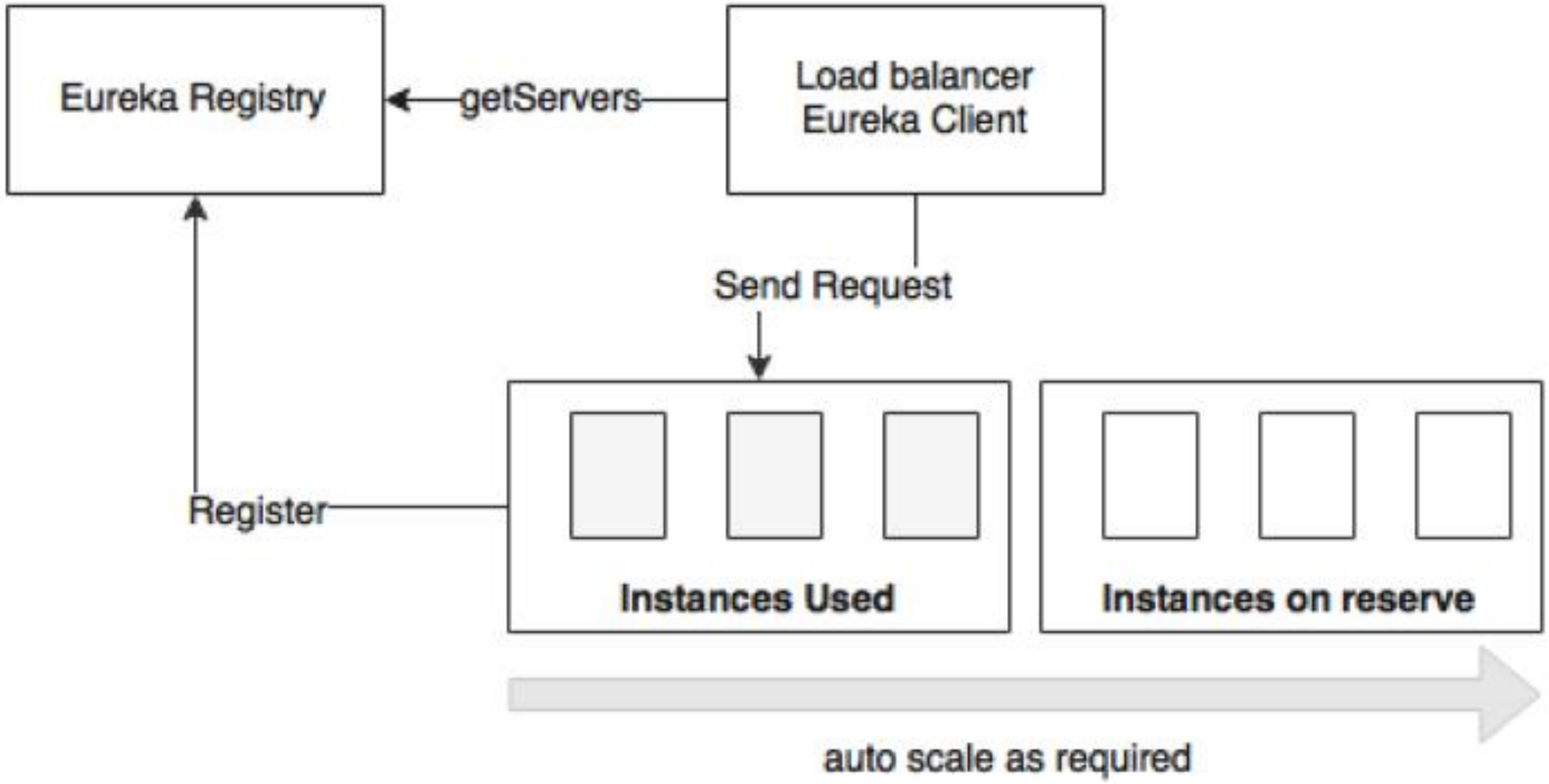
- The registry approach decouples the service instances.
- It also eliminates the need to manually maintain service addresses in the load balancer or configure virtual IPs:



# Understanding the concept of autoscaling

- Autoscaling is an approach to automatically scaling out instances based on the resource usage to meet the SLAs by replicating the services to be scaled.
- The system automatically detects an increase in traffic, spins up additional instances, and makes them available for traffic handling.
- Similarly, when the traffic volumes go down, the system automatically detects and reduces the number of instances by taking active instances back from the service:

# Understanding the concept of autoscaling



# The benefits of autoscaling

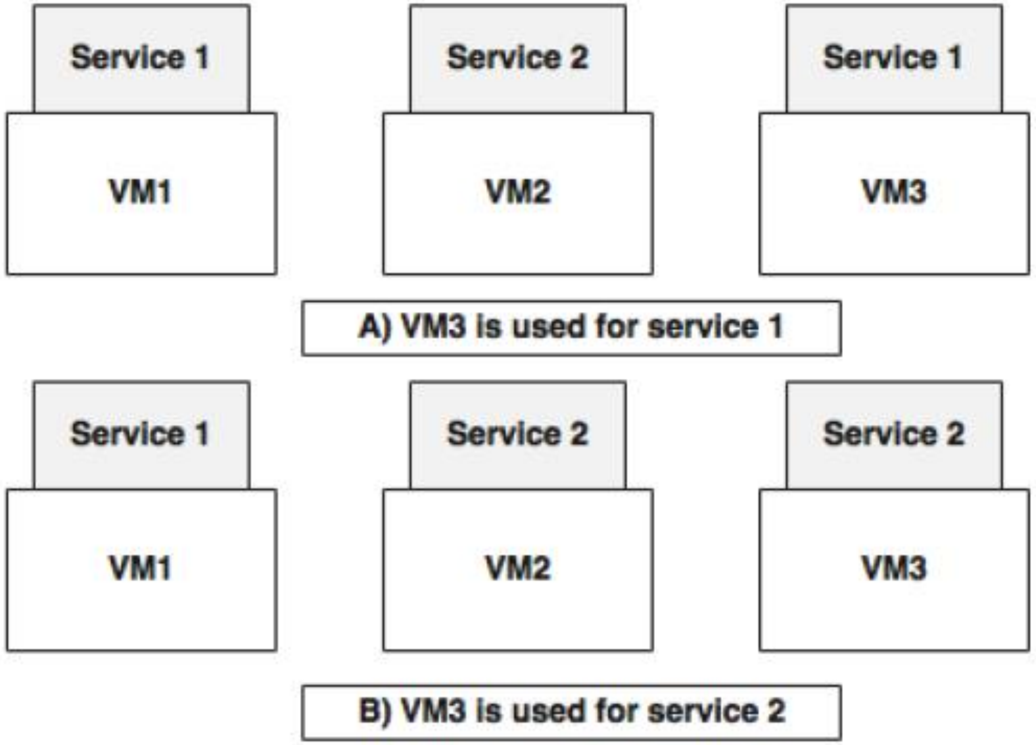
- It has high availability and is fault tolerant
- It increases scalability
- It has optimal usage and is cost saving
- It gives priority to certain services or group of services

# Different autoscaling models

- Autoscaling can be applied at the application level or at the infrastructure level.
- In a nutshell, application scaling is scaling by replicating application binaries only, whereas infrastructure scaling is replicating the entire virtual machine, including application binaries.

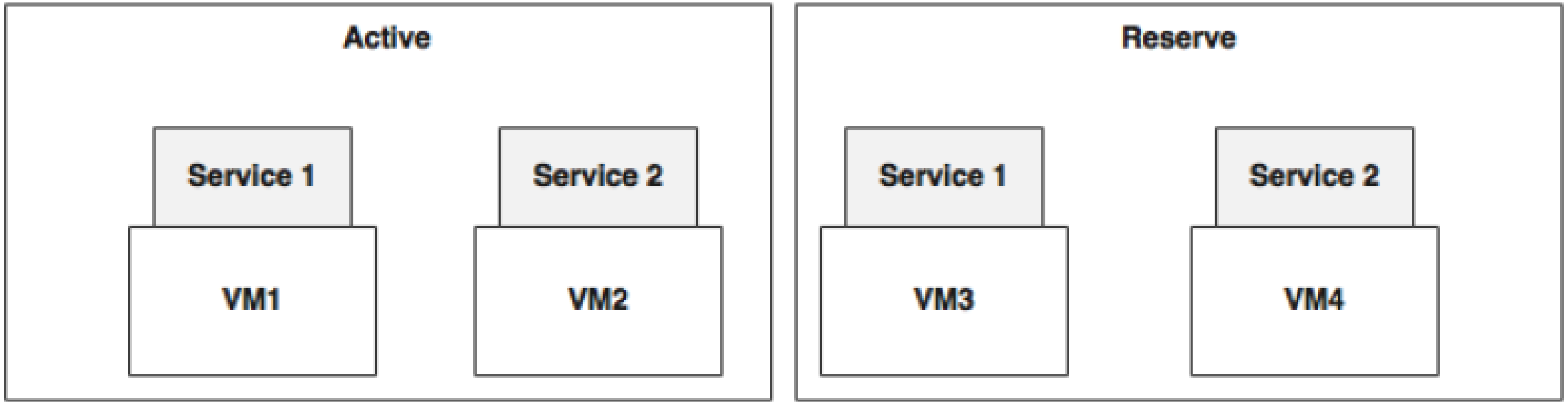
# Autoscaling an application

- In this scenario, scaling is done by replicating the microservices, not the underlying infrastructure, such as virtual machines.
- This gives flexibility in reusing the same virtual or physical machines for different services:



# Autoscaling an infrastructure

- In most cases, this will create a new VM on the fly or destroy the VMs based on the demand:



# Day - 6

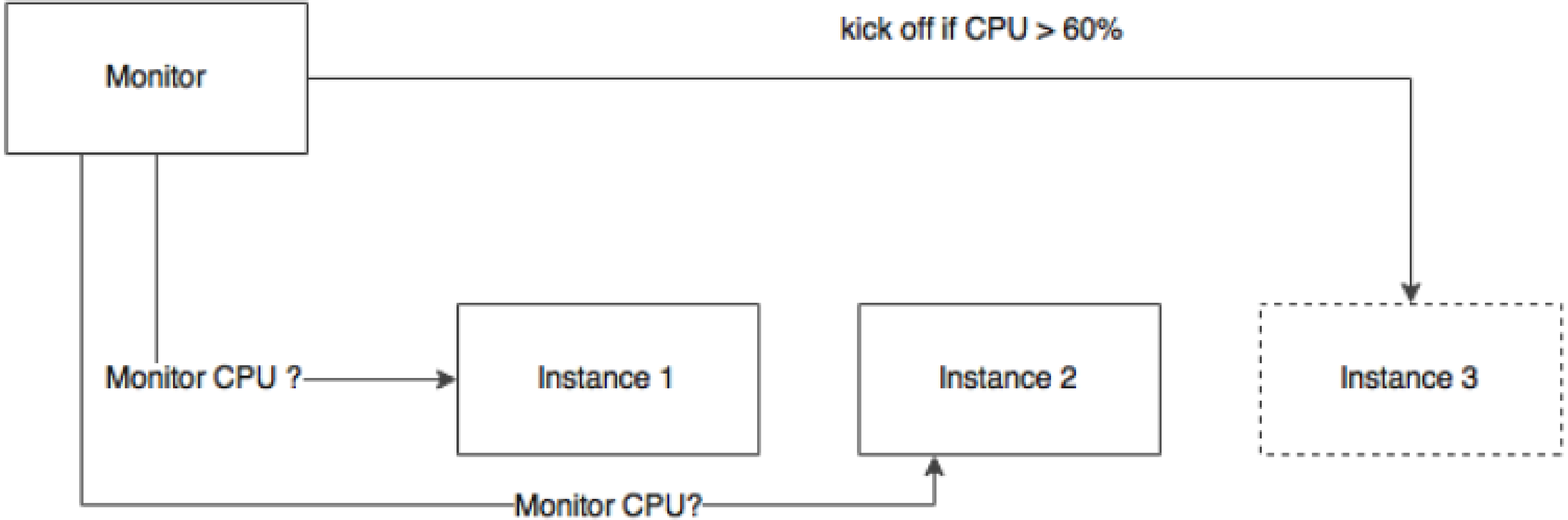
# Autoscaling Approaches

# Autoscaling approaches

- Autoscaling is handled by considering different parameters and thresholds.
- In this section, we will discuss the different approaches and policies that are typically applied to take decisions on when to scale up or down.

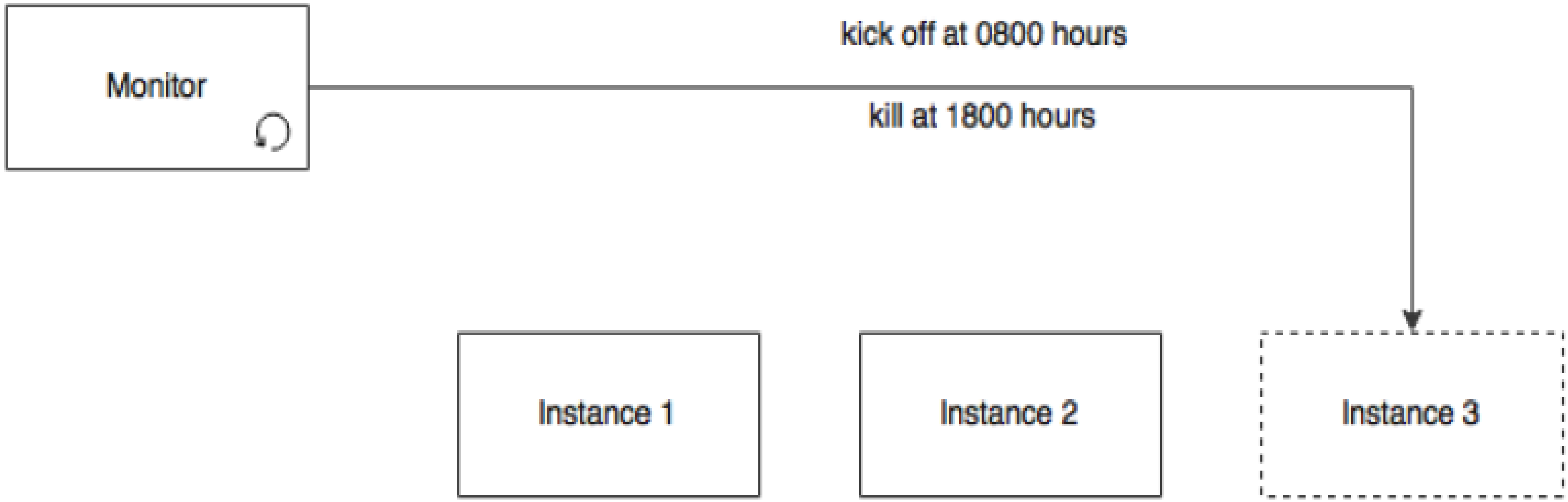
# 1. Scaling with resource constraints

- This approach is based on real-time service metrics collected through monitoring mechanisms.



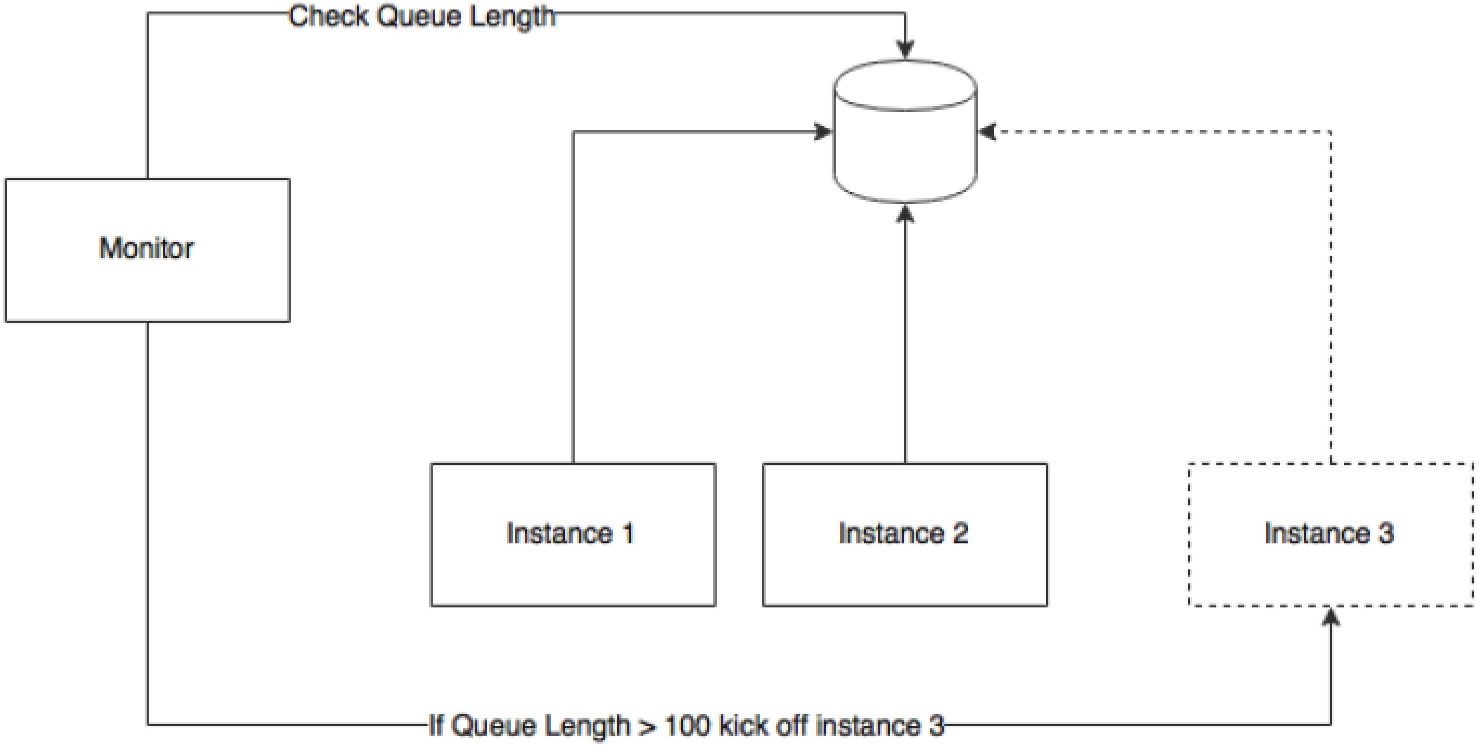
# 2. Scaling during specific time periods

- Time-based scaling is an approach to scaling services based on certain periods of the day, month, or year to handle seasonal or business peaks.



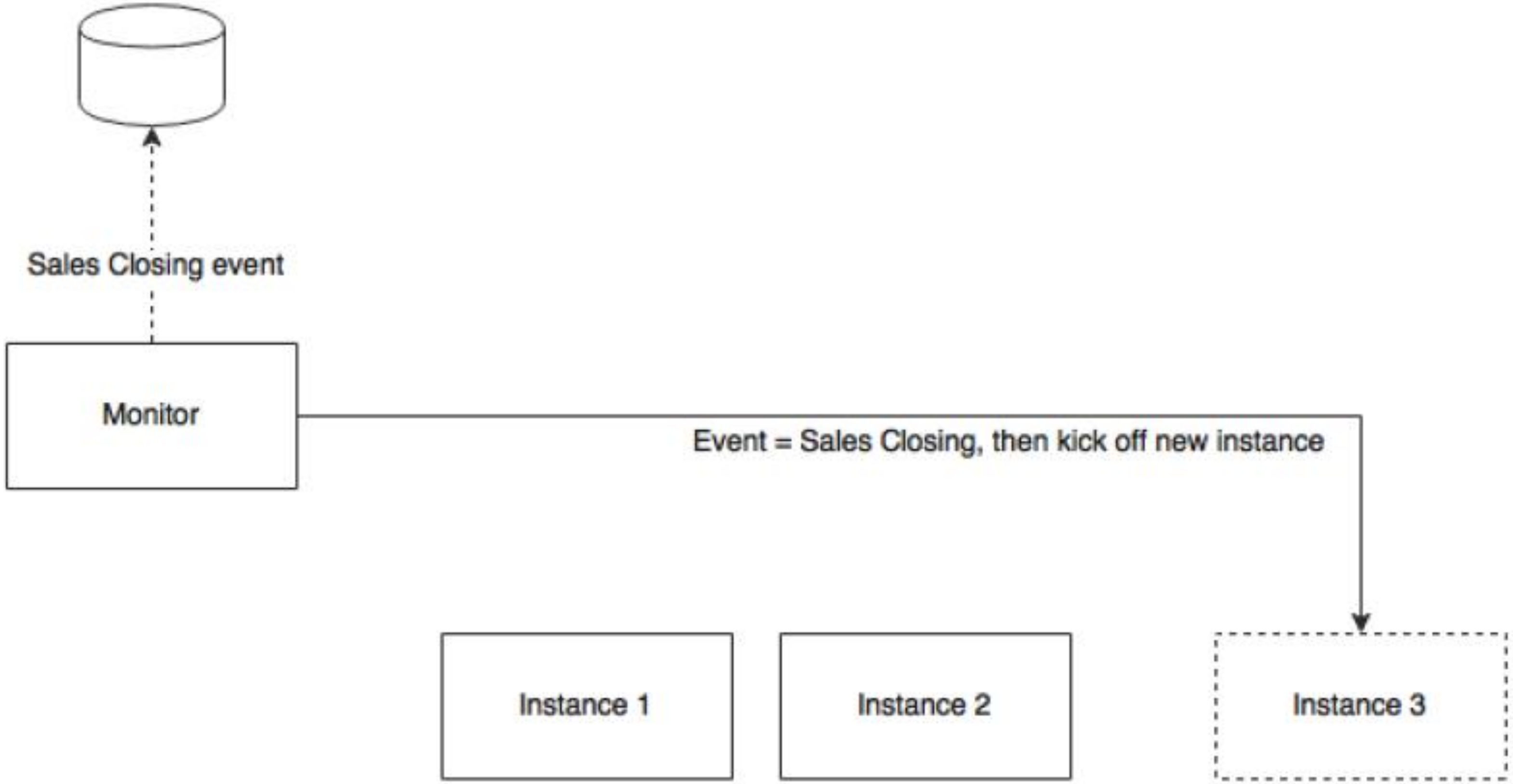
# 3. Scaling based on the message queue length

- This is particularly useful when the microservices are based on asynchronous messaging.
- In this approach, new consumers are automatically added when the messages in the queue go beyond certain limits:



# 4. Scaling based on business parameters

- In this case, adding instances is based on certain business parameters



# 5. Predictive autoscaling

- Predictive scaling is a new paradigm of autoscaling that is different from the traditional real-time metrics-based autoscaling.
- A prediction engine will take multiple inputs, such as historical information, current trends, and so on, to predict possible traffic patterns.
- Autoscaling is done based on these predictions.
- Predictive autoscaling helps avoid hardcoded rules and time windows.
- Instead, the system can automatically predict such time windows.
- In more sophisticated deployments, predictive analysis may use **cognitive computing mechanisms** to predict autoscaling.

# 5. Predictive autoscaling

- In the cases of sudden traffic spikes, traditional autoscaling may not help. Before the autoscaling component can react to the situation, the spike would have hit and damaged the system. The predictive system can understand these scenarios and predict them before their actual occurrence.  An example will be handling a flood of requests immediately after a planned outage.
- **Netflix Scryer** is an example of such a system that can predict resource requirements in advance.

# Recap of Day – 6

- What is API Gateway?
- Spring cloud routing – Zuul
- Setting up Zuul

- Autoscaling microservices
- Scaling microservices with Spring Cloud
- Understanding the concept of autoscaling
- The benefits of autoscaling
- Different autoscaling models
- Autoscaling Approaches

# Recap of overall Agenda

- Use Spring Boot to build standalone web applications and RESTful services
- Understand the Configuration techniques that Spring Boot Provides
- Build Spring boot based Microservices for JSON and XML data exchange
- Monitor services using the Actuator
- Understand the major components of Netflix OSS
- Register services with a Eureka Service
- Implement “client” load balancing with Ribbon to Eureka managed Services
- Isolating from failures with Hystrix
- Filter requests to your Microservices using Zuul
- Define Feign clients to your services
- Scaling Microservices with Spring Cloud

# Additional Recommended Reading

- Spring Microservices in Action
- Spring Boot Intermediate Microservices
- Enterprise Java Microservices

Mphasis
The Next Applied

# THANK YOU

## About Mphasis

Mphasis (BSE: 526299; NSE: MPHASIS) applies next-generation technology to help enterprises transform businesses globally. Customer centricity is foundational to Mphasis and is reflected in the Mphasis' Front2Back™ Transformation approach. Front2Back™ uses the exponential power of cloud and cognitive to provide hyper-personalized ($C=X2C^2_{TM}=1$) digital experience to clients and their end customers. Mphasis' Service Transformation approach helps 'shrink the core' through the application of digital technologies across legacy environments within an enterprise, enabling businesses to stay ahead in a changing world. Mphasis' core reference architectures and tools, speed and innovation with domain expertise and specialization are key to building strong relationships with marquee clients. Click here to know

## Important Confidentiality Notice

# Any Questions?

Manpreet.Bindra@mphasis.com

FOLLOW US IN THE LINK BELOW
@Mphasis