# Microservices infrastructure

August, 3-4nd 2019

Sergey Morenets, 2019


DEVELOPER 14 YEARS

TRAINER 6 YEARS

WRITER 4 BOOKS


# FOUNDER

ITSimulator



# SPEAKER


JAVA DAY
MINSK 2013

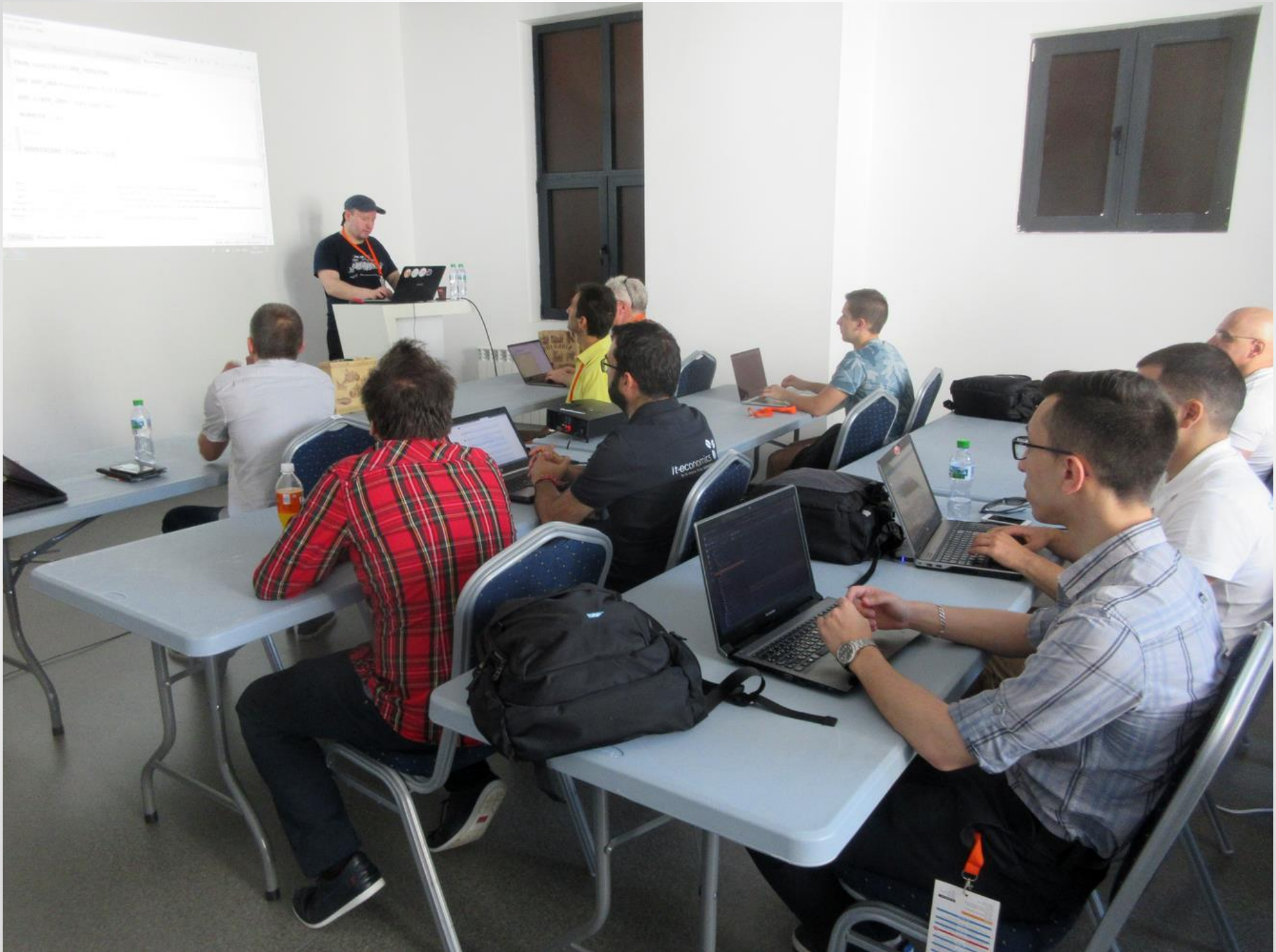

Dev(Talks):



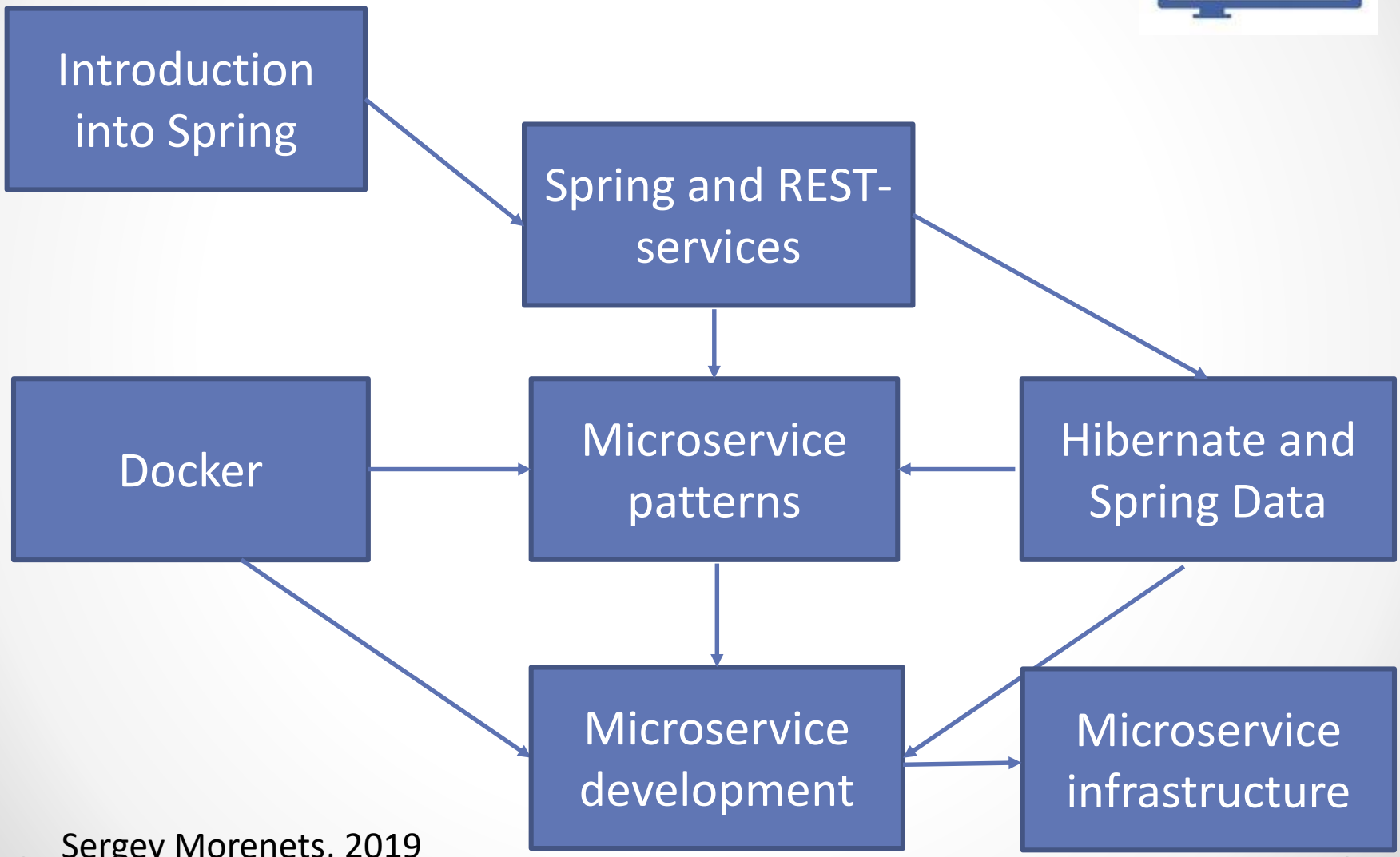


Java
frameworks
day

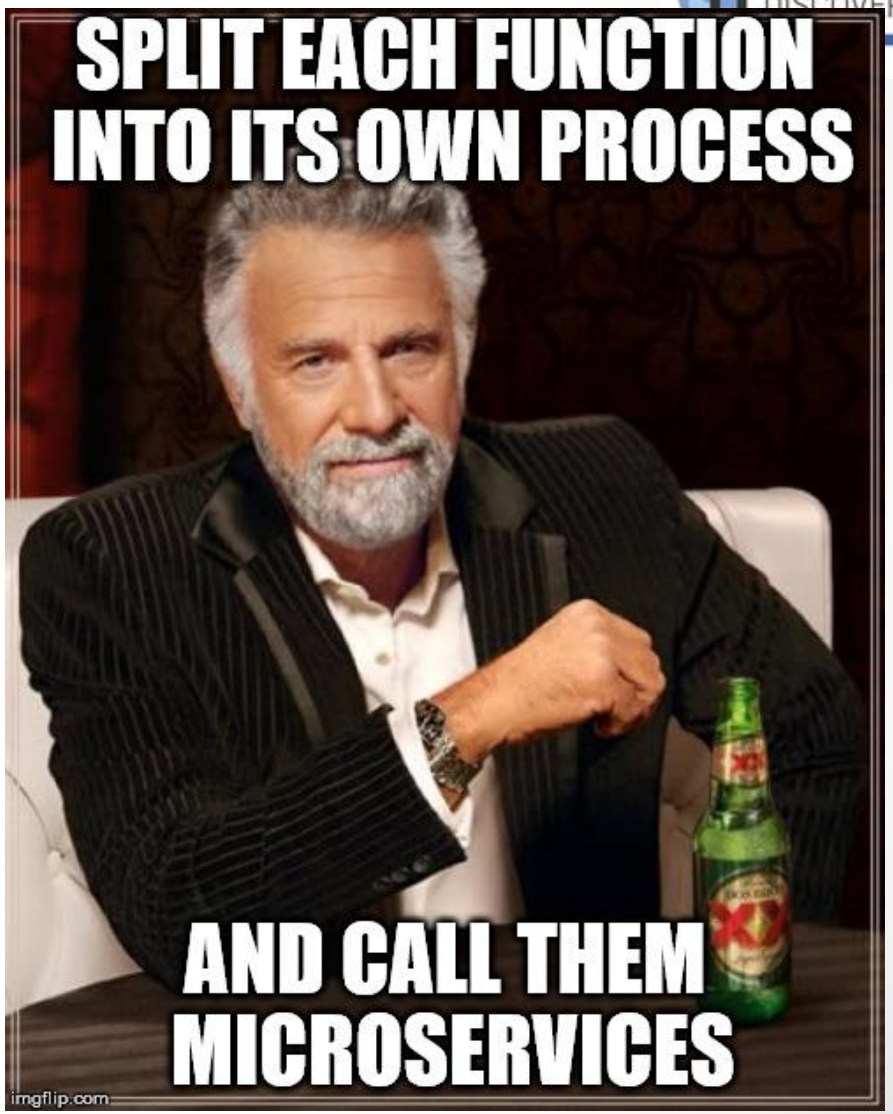

@sergey-morenets
DEVOXX
POLAND


it-economics

# Traning roadmap



# Agenda



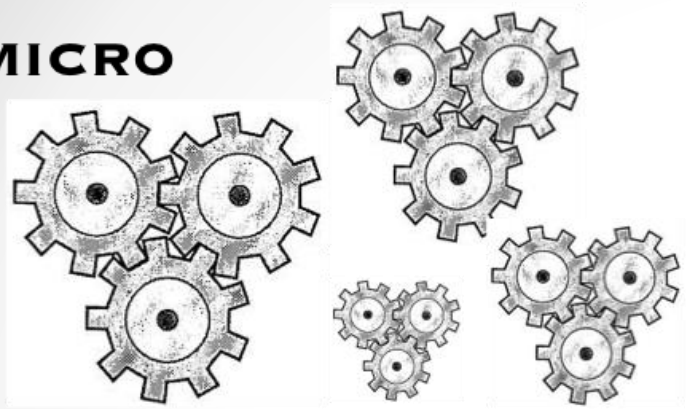

- Sergey Morenets, 2019

# Agenda


- ✓ Complexity of monolith applications
- ✓ Micro-service architecture. Pro and cons. Patterns
- ✓ Spring Cloud platform
- ✓ Configuration management
- ✓ Service discovery (Eureka, Consul, Zookeeper)
- ✓ Load balancing
- ✓ Resilience (Hystrix)
- ✓ Distributed tracing (Zipkin, Sleuth)
- ✓ Security (Vault)
- ✓ API Gateway

Sergey Morenets, 2019

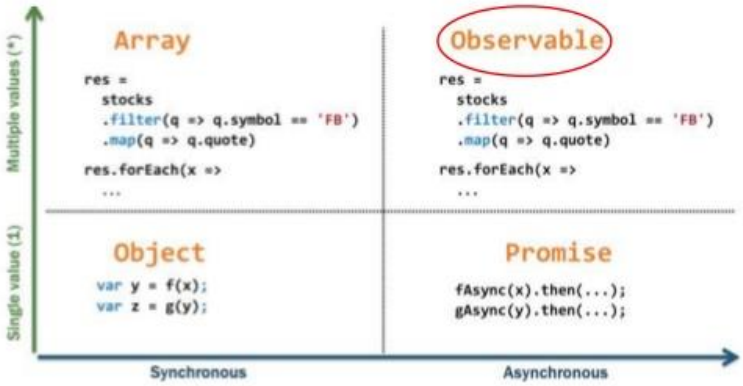
MICRO



SERVICES

IT DISCOVERY

## Reactive Programming




[5]

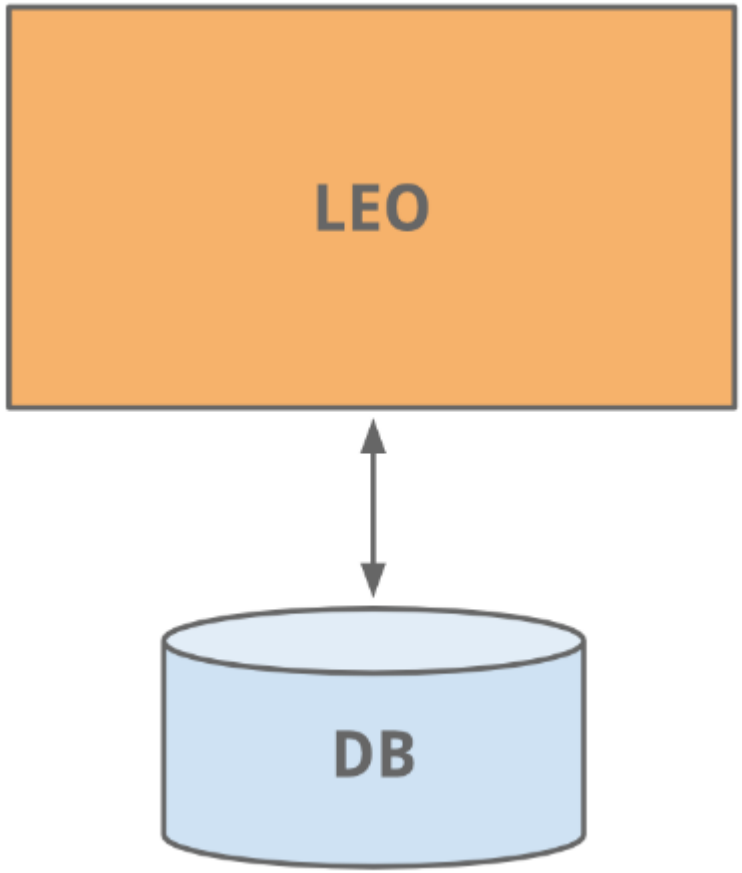


- Sergey Morenets, 2019

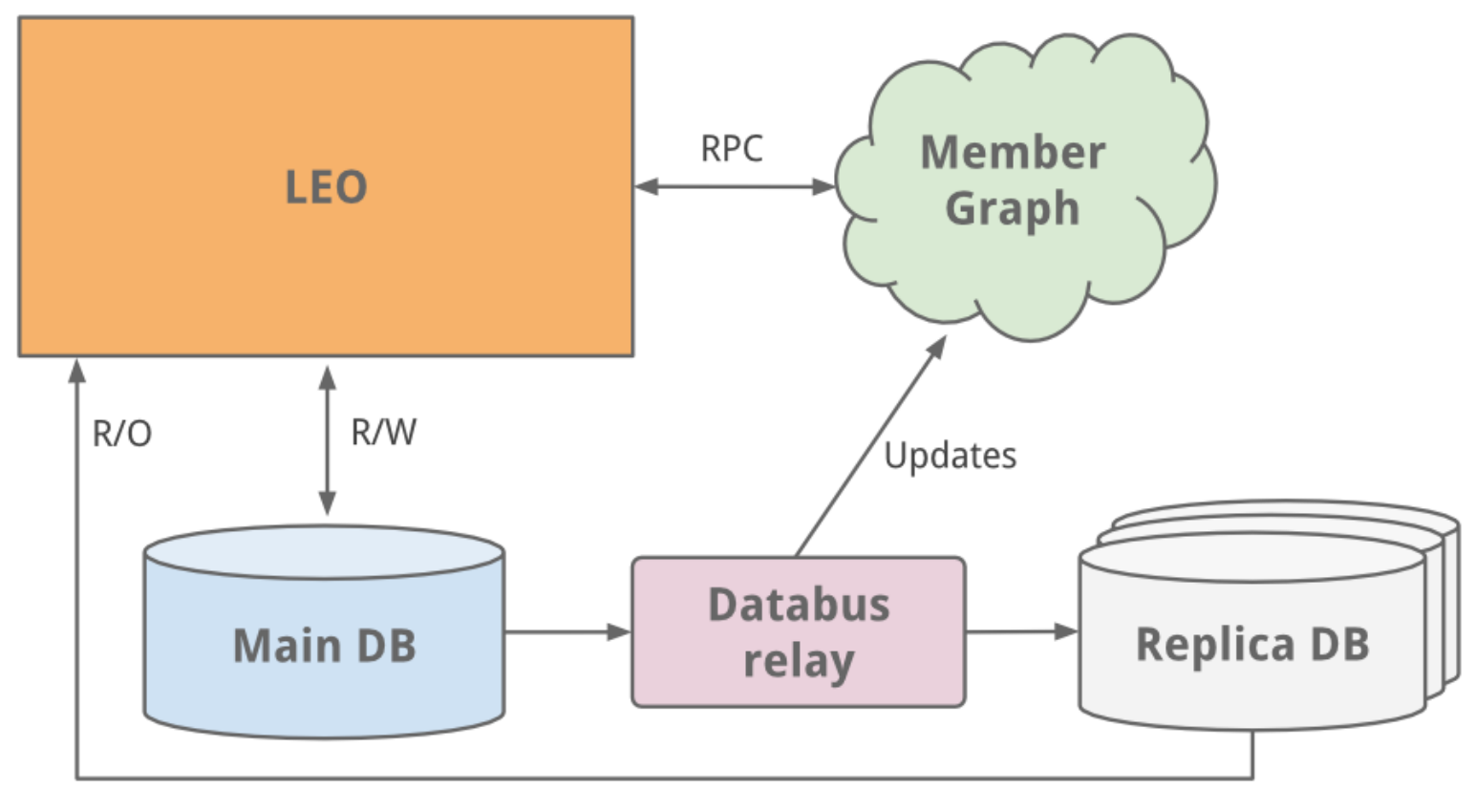
# Story with happy-end

Sergey Morenets, 2019

# LinkedIn. Long ago …



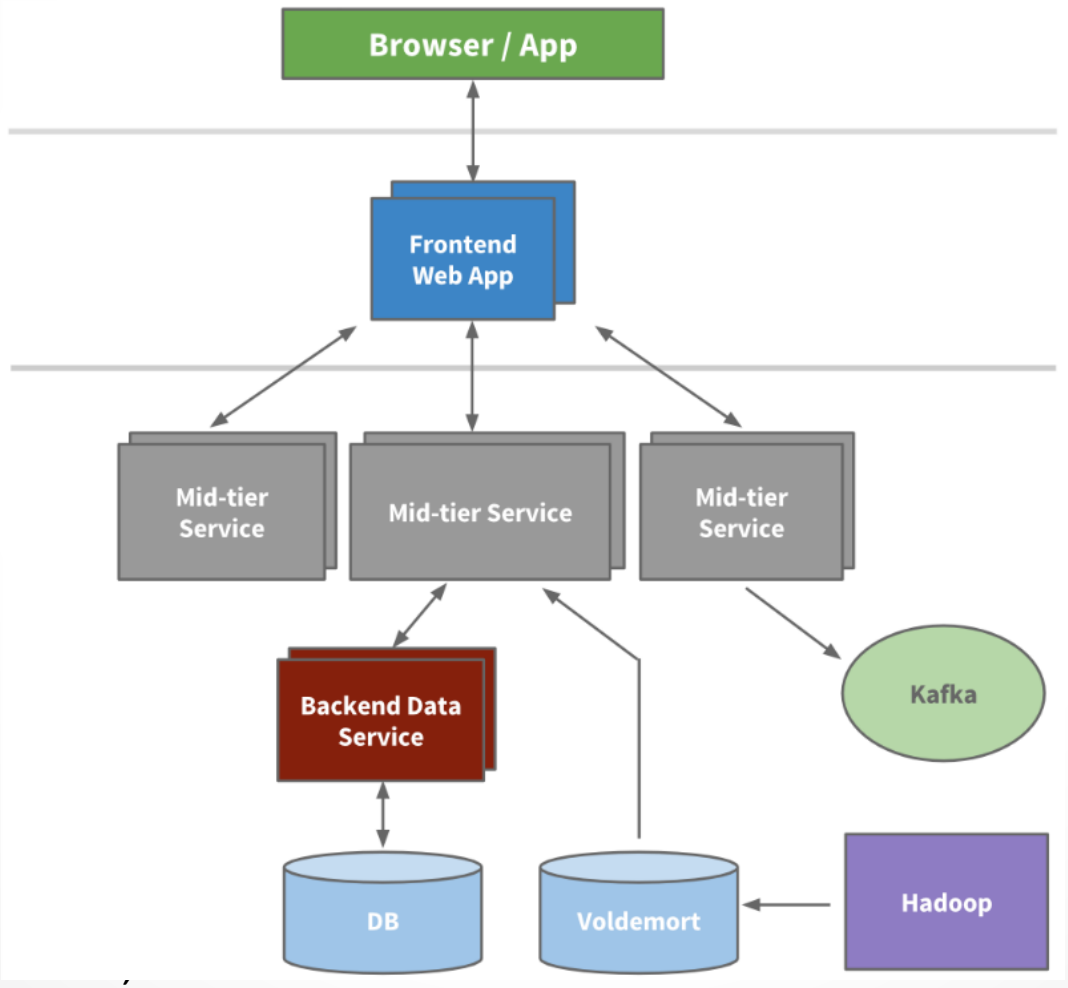
# LinkedIn. Some time in the past




Sergey Morenets, 2019

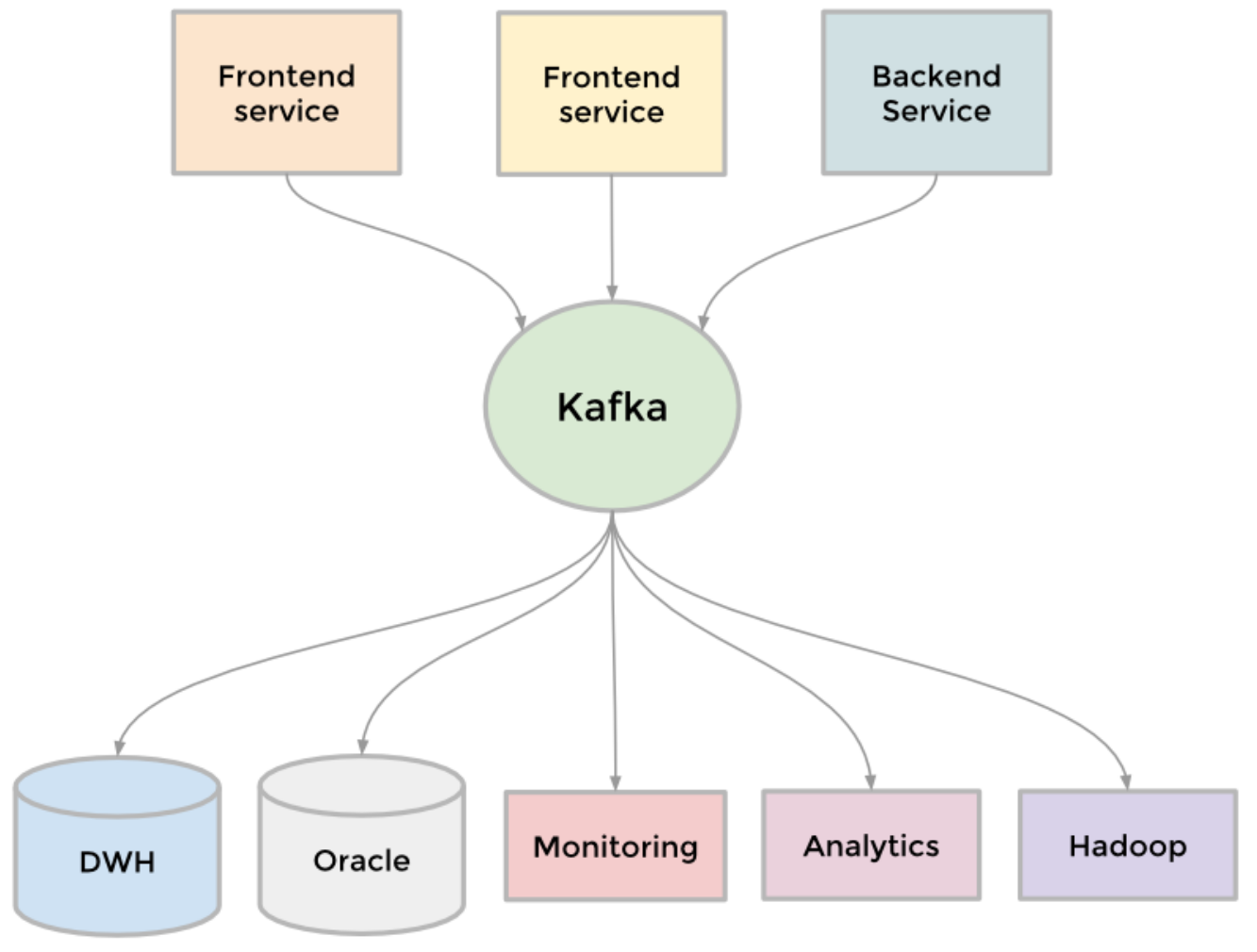
# LinkedIn. Some time in the past

# LinkedIn. Nowadays




- Sergey Mo

# LinkedIn. Nowadays




- Ser

# LinkedIn. Statistics (2016)


- More than 900 services
- 1.4 trillion messages (650 terabytes) per day
- More than 450 million users
- Peak 13 million messages (2.75 gigabytes) per second

# YouTube. Statistics (2017)


- 1 300 000 000 users
- 300 hours of video are uploaded every minute
- 5 000 000 000 videos are watched every single day
- 1 000 000 000 videos are watched in mobile every single day
- 30 000 000 visitors per day

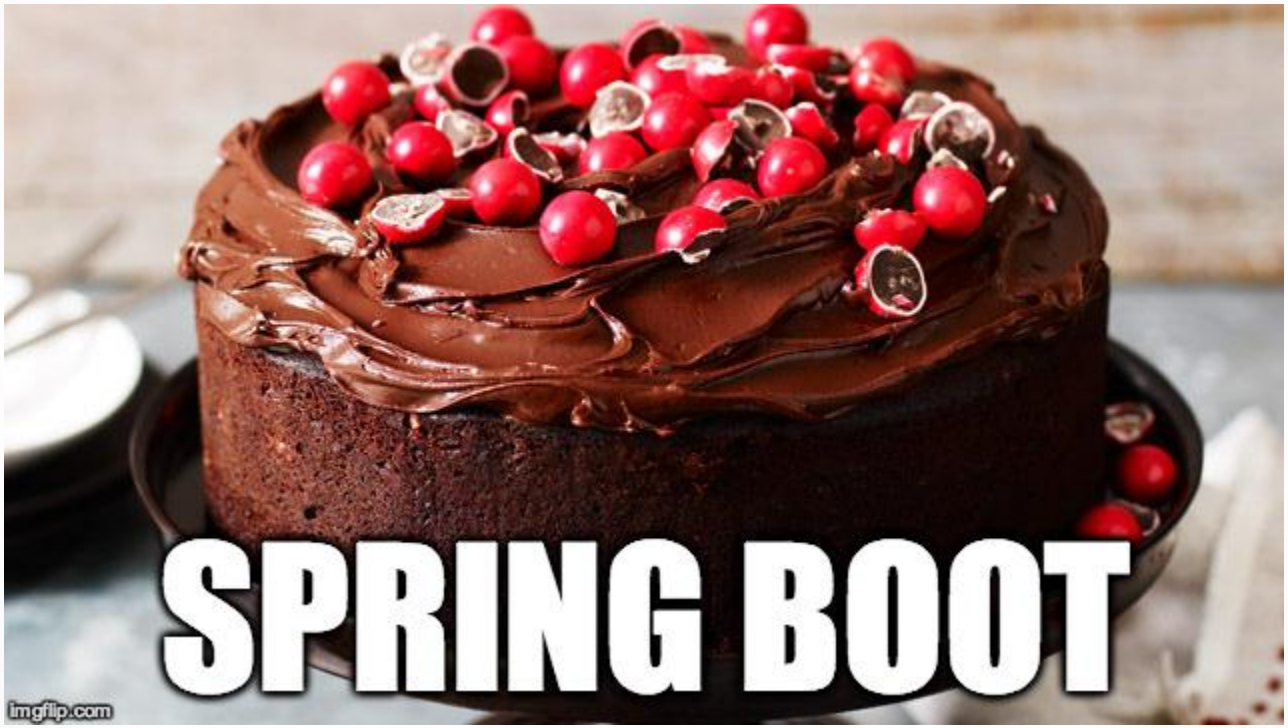



- Sergey Morenets, 2019

# Spring Framework


- Based on **DI**(IoC) conversion
- Integrated with most popular frameworks
- Contains over **30** sub-projects


Phil Webb
@phillip_webb
Читать
What would you like to complain about?
[ ] Too much magic
[ ] Too much boilerplate
Язык твита: английский
16:16 - 4 мар. 2016 г.
1 252 ретвита 1 289 отметок «Нравится»
116
1,3 тыс.
1,3 тыс.




Sergey Morenets, 2019

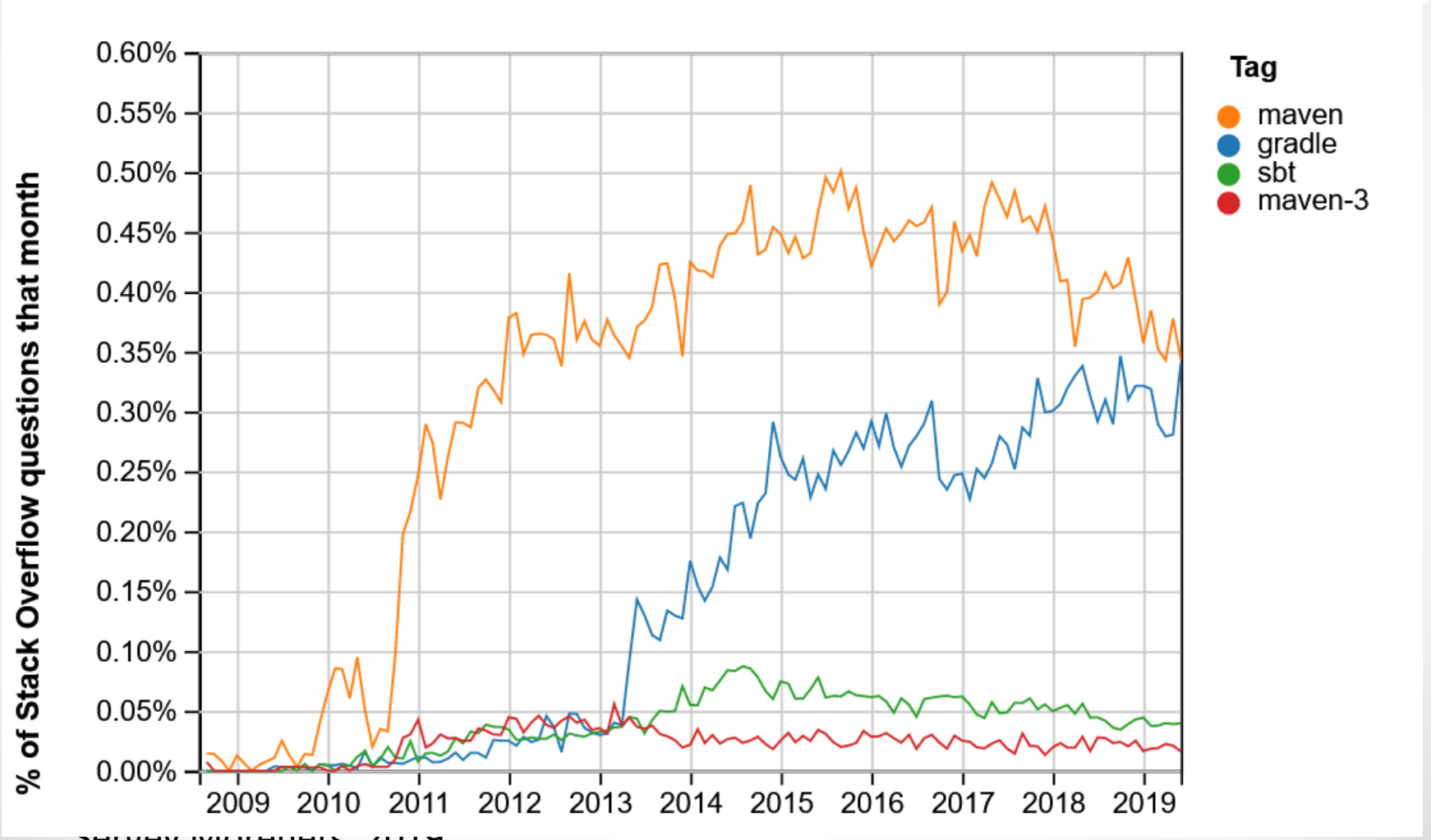
# Spring Boot


- Stand-alone Spring applications
- Embed Tomcat, Jetty, Undertow or Netty
- Automatically Spring configuration
- Convention-over-configuration
- Zero XML configuration
- POM starters
- Actuator
- Dev tools

# Build management

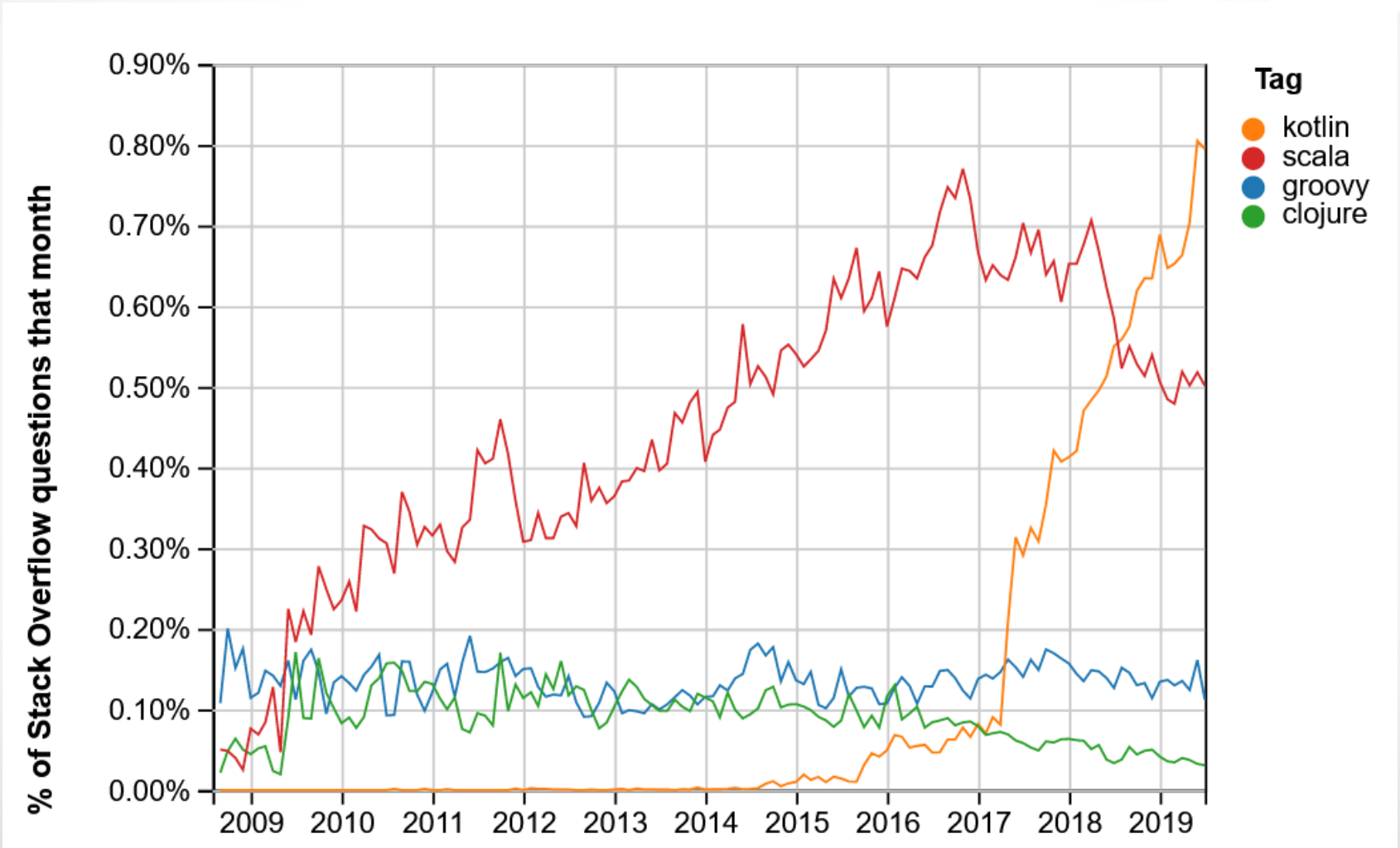



# Maven vs Gradle vs Sbt




Sergey Morenets, 2019

# Groovy vs Kotlin




- Sergey Morenets, 2019

# REST service


```
@RestController
@RequestMapping("book")
public class BookController {

    @GetMapping("/{id}")
    public Book getBook(@PathVariable int id) {
        return bookRepository.findBookById(id);
    }
```

# IDE

# Built management tasks


**Maven**

- spring-boot:run
- spring-boot:repackage

**Gradle**

- bootRun
- bootRepackage

# Spring Boot. Dev tools


- ✓ Automatic restart when file(s) on a classpath changes
- ✓ **LiveReload** server support
- ✓ Remote application support
- ✓ Dev customization by default

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <version>${spring.boot.version}</version>
    <optional>true</optional>
</dependency>
```

# Dependency management. Maven


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

```
<properties>
    <activemq.version>5.15.3</activemq.version>
    <antlr2.version>2.7.7</antlr2.version>
    <appengine-sdk.version>1.9.62</appengine-sdk.vers
    <artemis.version>2.4.0</artemis.version>
    <aspectj.version>1.8.13</aspectj.version>
    <assertj.version>3.9.1</assertj.version>
    <atomikos.version>4.0.6</atomikos.version>
    <bitronix.version>2.1.4</bitronix.version>
    <build-helper-maven-plugin.version>3.0.0</build-h
    <byte-buddy.version>1.7.10</byte-buddy.version>
    <caffeine.version>2.6.2</caffeine.version>
```

**spring-boot-dependencies.pom.xml**

# What is monolith application?

Sergey Morenets, 2019

# Monolith application


- ✓ Single **deployment** unit(WAR, EAR)
- ✓ Single codebase
- ✓ No restrictions on the **project size**
- ✓ Single **database** (RDBMS)
- ✓ Single **language**
- ✓ Long development **iterations**
- ✓ Fixed technology **stack**(JEE, Spring, Hibernate)
- ✓ **ACID** principle
- ✓ One or few **teams**

# Issues

Sergey Morenets, 2019

# Issues


- ✓ Hard to maintain
- ✓ Hard to add new features (**fast**)
- ✓ Hard to scale (specific components)
- ✓ Hard to deploy
- ✓ Slower to start
- ✓ Slower to work in IDE
- ✓ Cannot deploy single module
- ✓ Cannot learn the whole project

# Transition

Monolith → Microservices

# Transition


Monolith → Microservices

Development
Deployment
Testing
Maintaining

# Task 1. Monolith application

1. Import **microservices** project into your IDE and review **monolith** project.
2. Review project functionality and domain model.
3. Update **ShopController** class and add necessary **Spring** annotations.
4. Run application as **Spring Boot** project and observe its behavior.
5. Identify issues related by the monolith ar
   possible solutions for them.

# What is micro-service?

Sergey Morenets, 2019

# Micro-service

- ✓ Loosely coupled service oriented architecture with bounded contexts

- ✓ Small autonomous service

Sergey Morenets, 2019

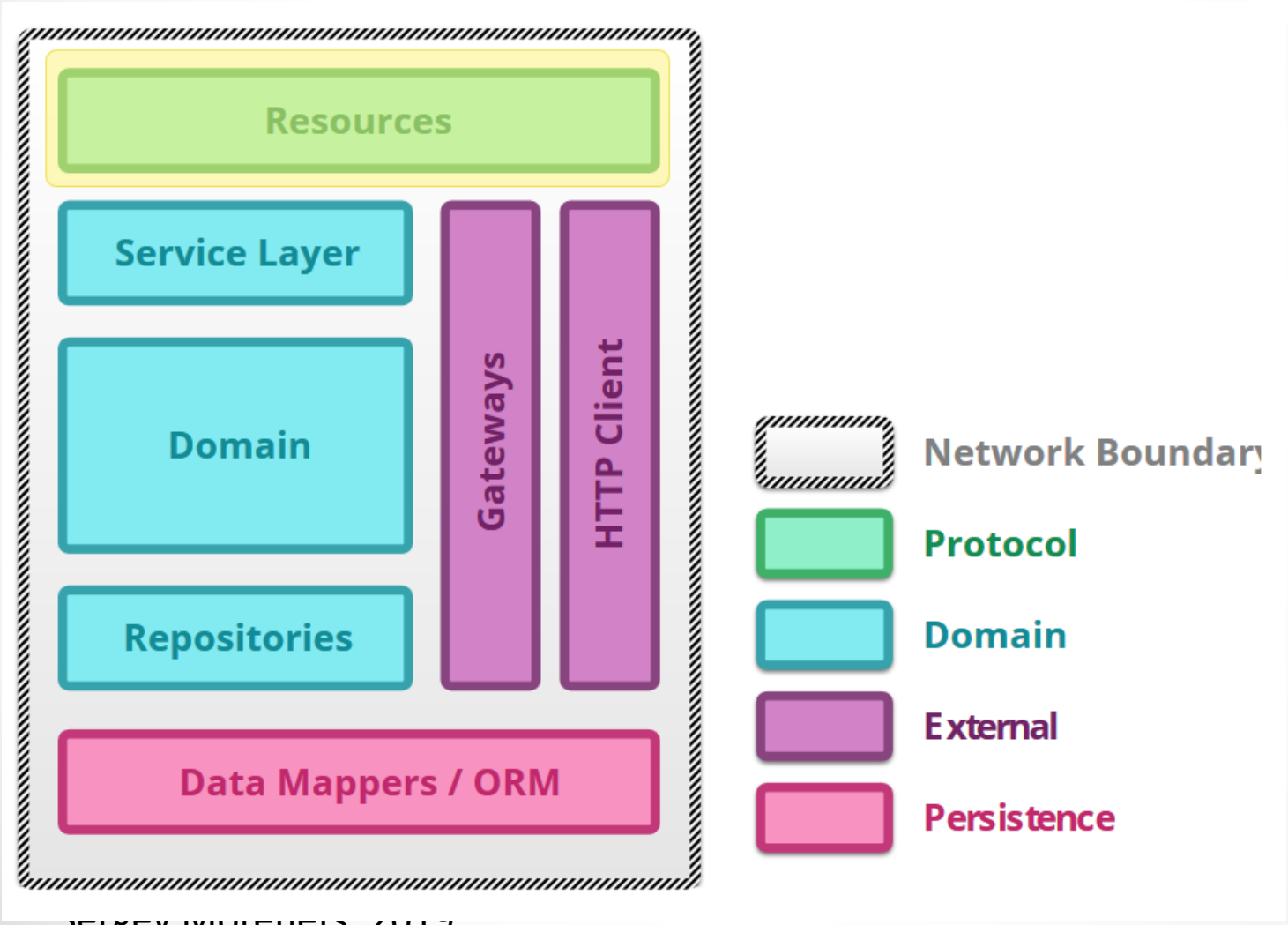
# Micro-services


- ✓ **Separately** written, deployed, scaled and maintained
- ✓ **Independently** upgraded
- ✓ **Easy** to understand/document
- ✓ Provides **business** features
- ✓ **Fast** deployment
- ✓ Use **cutting-edge** deployment
- ✓ Resolve **resource conflicts**(CPU, memory)
- ✓ Communication via **lightweight** protocols/formats

# Micro-services

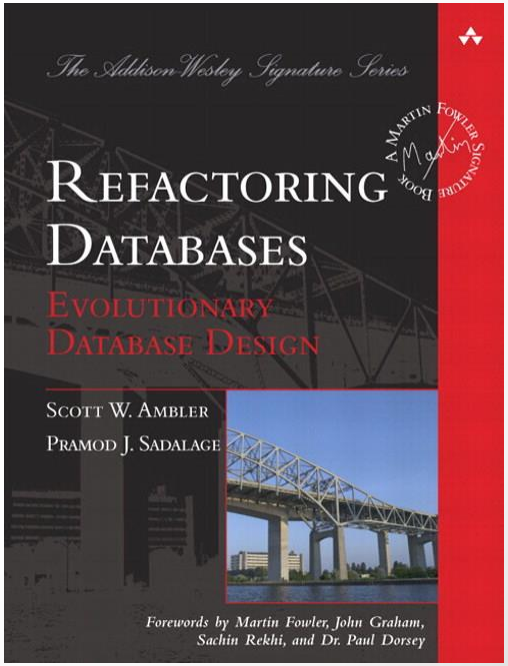

- ✓ **Smaller** and simpler applications
- ✓ Fewer **dependencies** between components
- ✓ Scale and develop independently
- ✓ Easy to introduce new **technologies**
- ✓ Cost, size and risk of changes reduced
- ✓ Easy to **test** single functionality
- ✓ Easy to introduce **versioning**
- ✓ **Cross-functional** distributed teams
- ✓ Improved security due to multiple data-sources
- ✓ Increased uptime

# Microservice structure



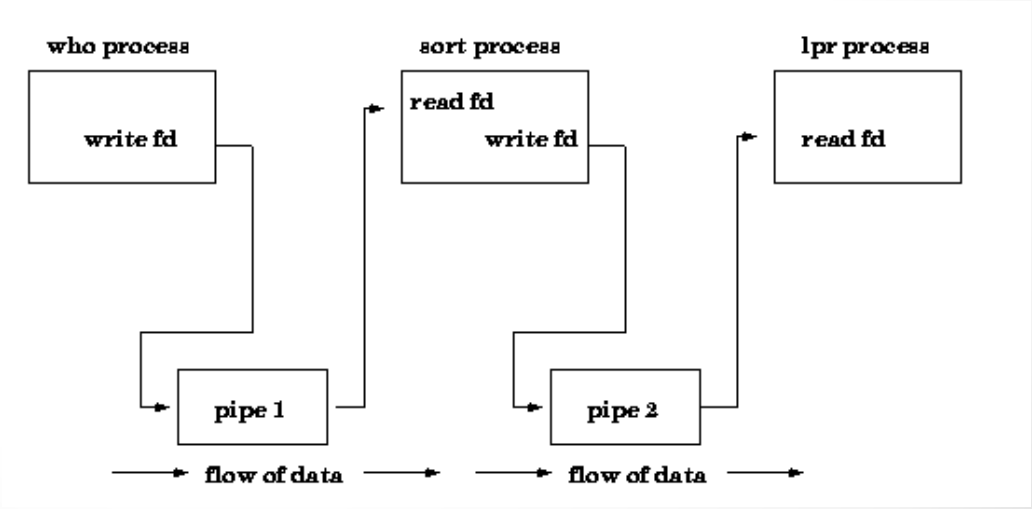
# Transition

- ✓ Split domain model/**business logic**
- ✓ Split data model/persistence layer
- ✓ Split **database** (including DB migration)
- ✓ Split/introduce **technologies**
- ✓ Split **API** (optional)
- ✓ Split tests and add **new** integration tests
- ✓ Split into **deployable modules**



# Partitioning strategies

- By entity(Customer – Product – Order)
- By use case(Book – Buy – Search)
- Single Responsibility Principle
- By datasource
- Bounded context



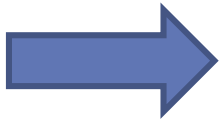
# Enterprise model. Client


- ✓ Identifier
- ✓ Name
- ✓ Address
- ✓ Email/Phone
- ✓ Credit card number
- ✓ Expiration date
- ✓ Discount
- ✓ Purchases

# Splitting data model

Client. **Purchase service**

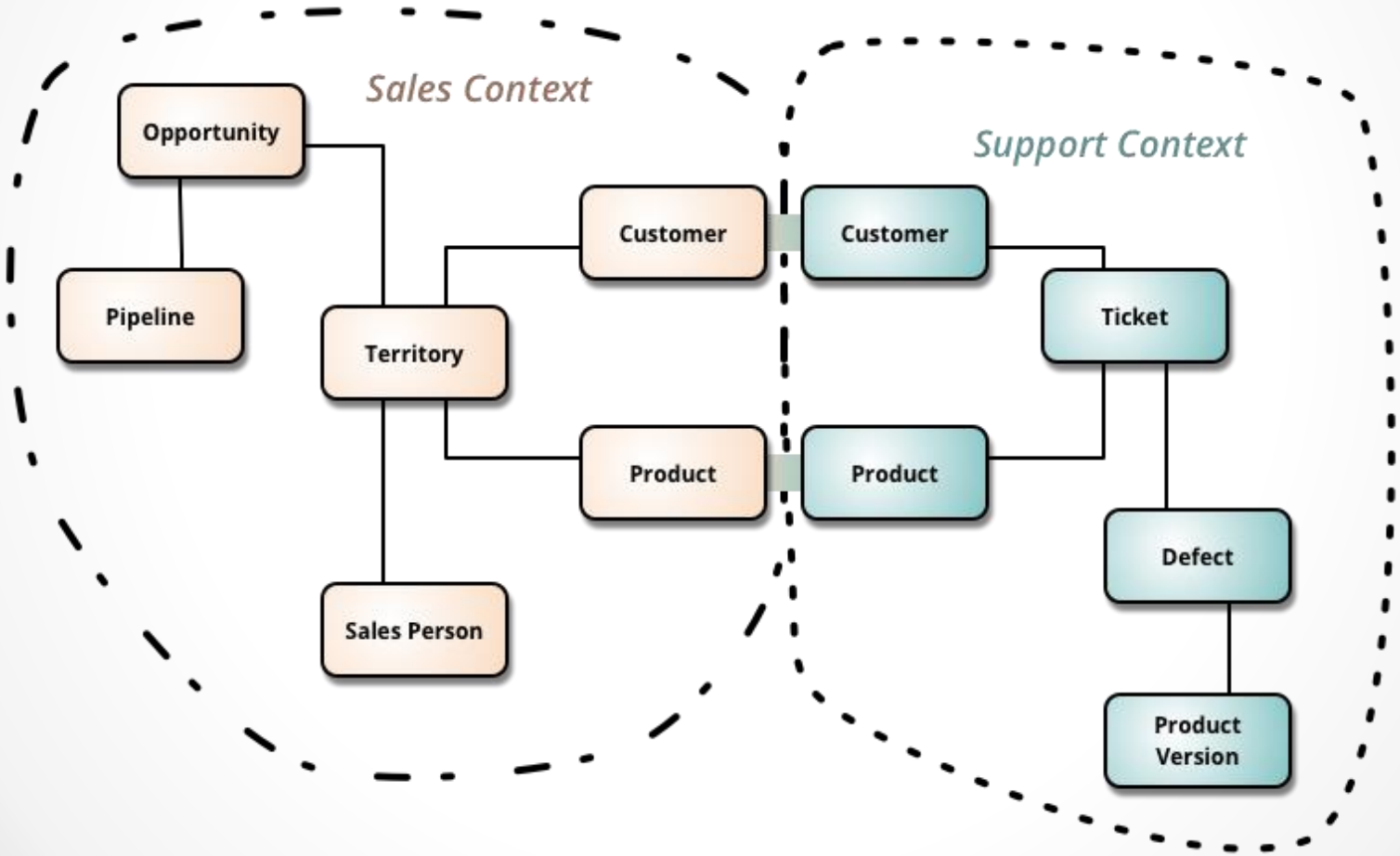
- ✓ Identifier
- ✓ Credit card number
- ✓ Expiration date
- ✓ Discount
- ✓ Purchases



Client. **Delivery service**

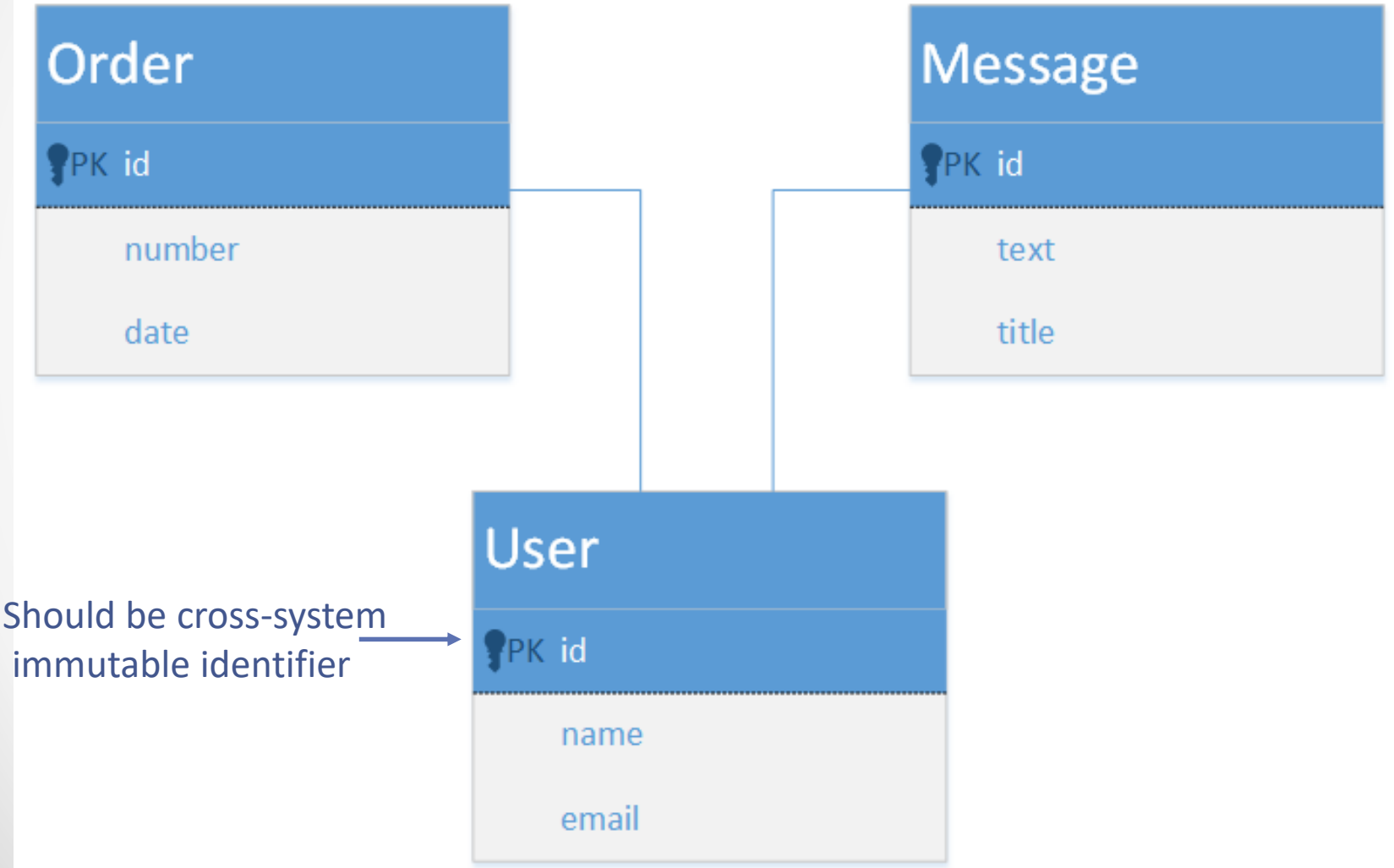
- ✓ Identifier
- ✓ Name
- ✓ Address
- ✓ Email/Phone

# Bounded context




Sergey Morenets, 2019

# Splitting data model



# Task 2. Splitting monolith


1. Review monolith application again. Try to convert it to micro-service architecture gradually.
2. Extract **functionality** that goes to the separate projects
3. Split **domain model**
4. Split **DAO layer(repository)**
5. Split **services**
6. Split **controllers**

# Questions

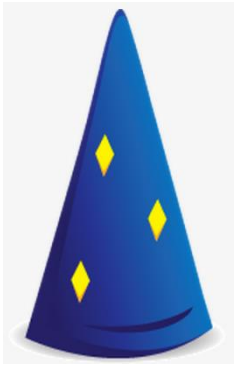

- ✓ Service discovery & lookup
- ✓ Load balancing
- ✓ Auto-scaling
- ✓ Protocols and data formats
- ✓ Security
- ✓ API management
- ✓ Configuration management
- ✓ Testing (functional, security, performance)
- ✓ Data access
- ✓ Monitoring

# Microservices


- ✓ API services
- ✓ Integration services (security, discovery, resilience, gateway, load balancer)

# Microservice frameworks


Full-stack



NETFLIX OSS

lagom

Microprofile


Open Liberty

helidon.io

Microframeworks

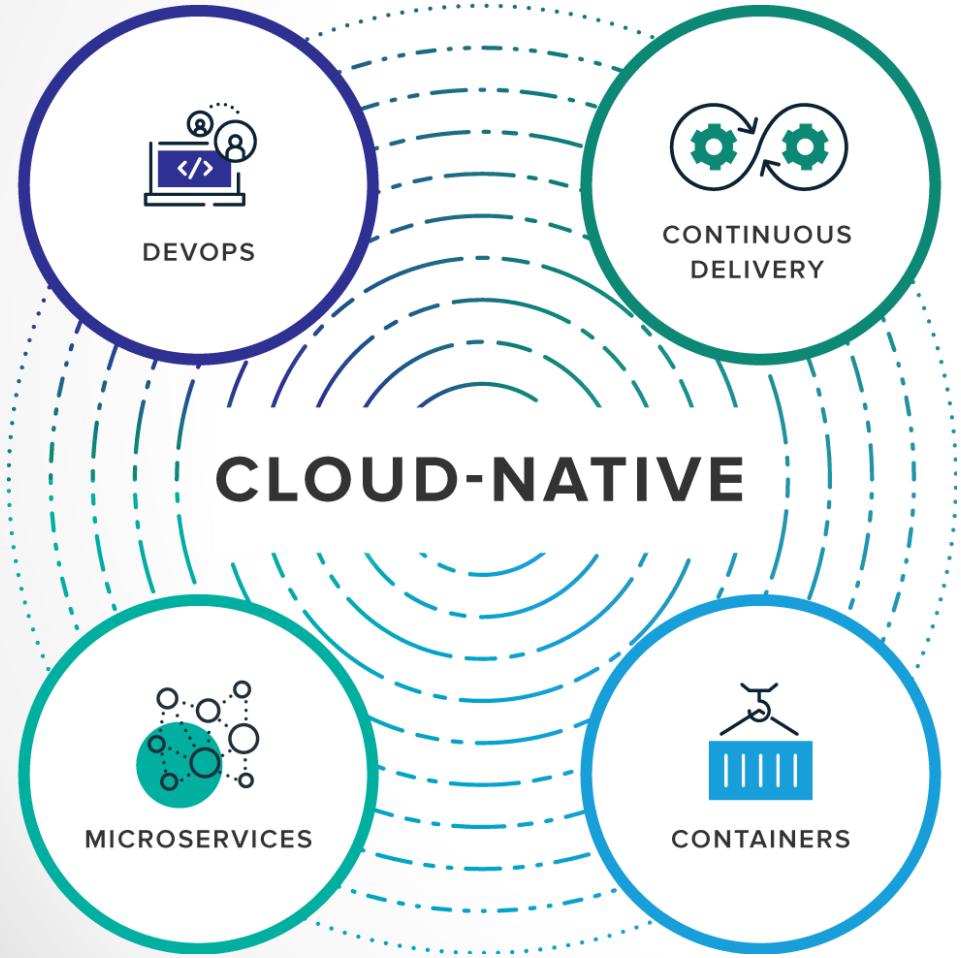
javalin

Spark

MICRONAUT

- Sergey Morenets, 2019

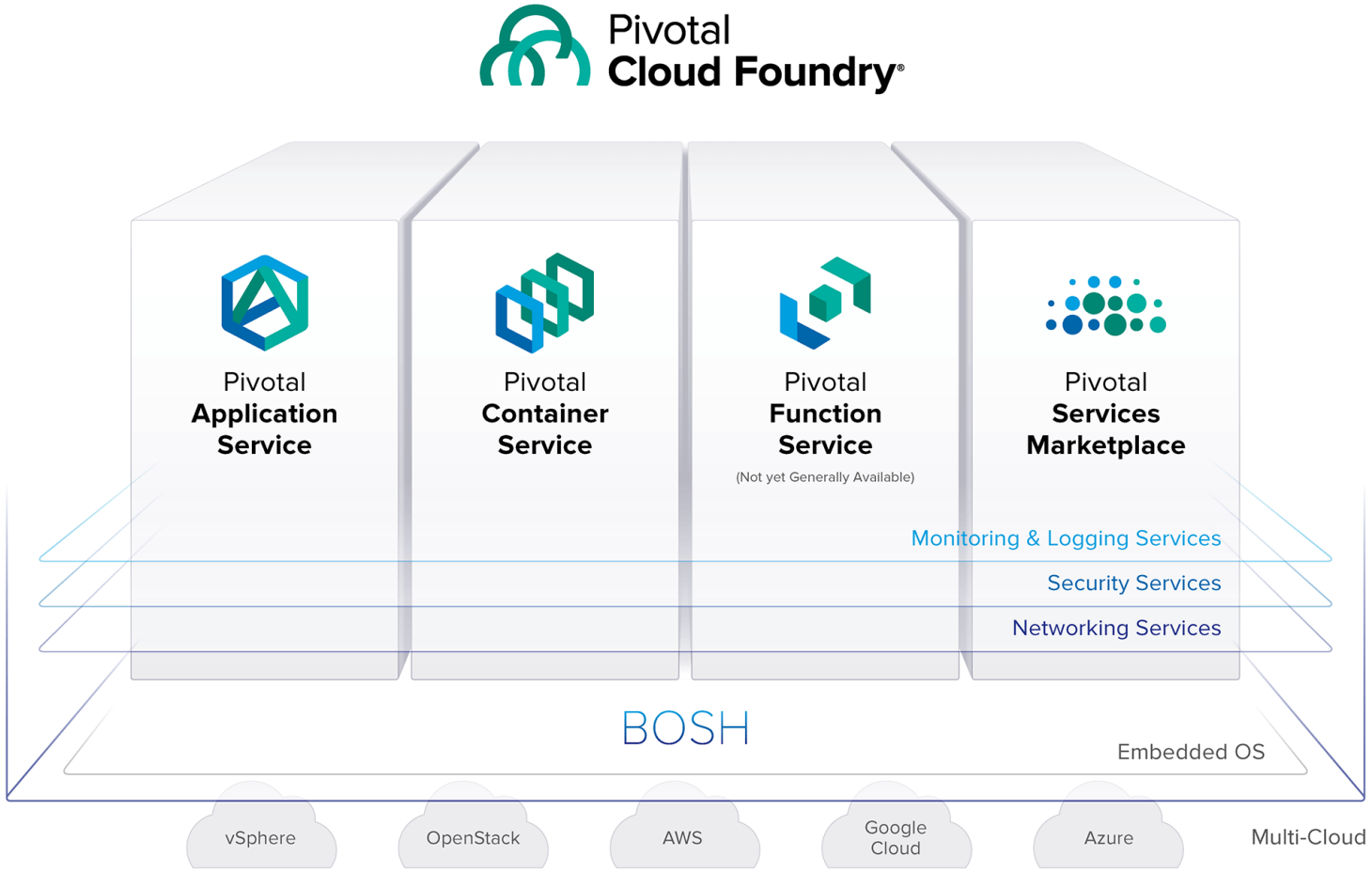
# Pivotal. Cloud-native applications

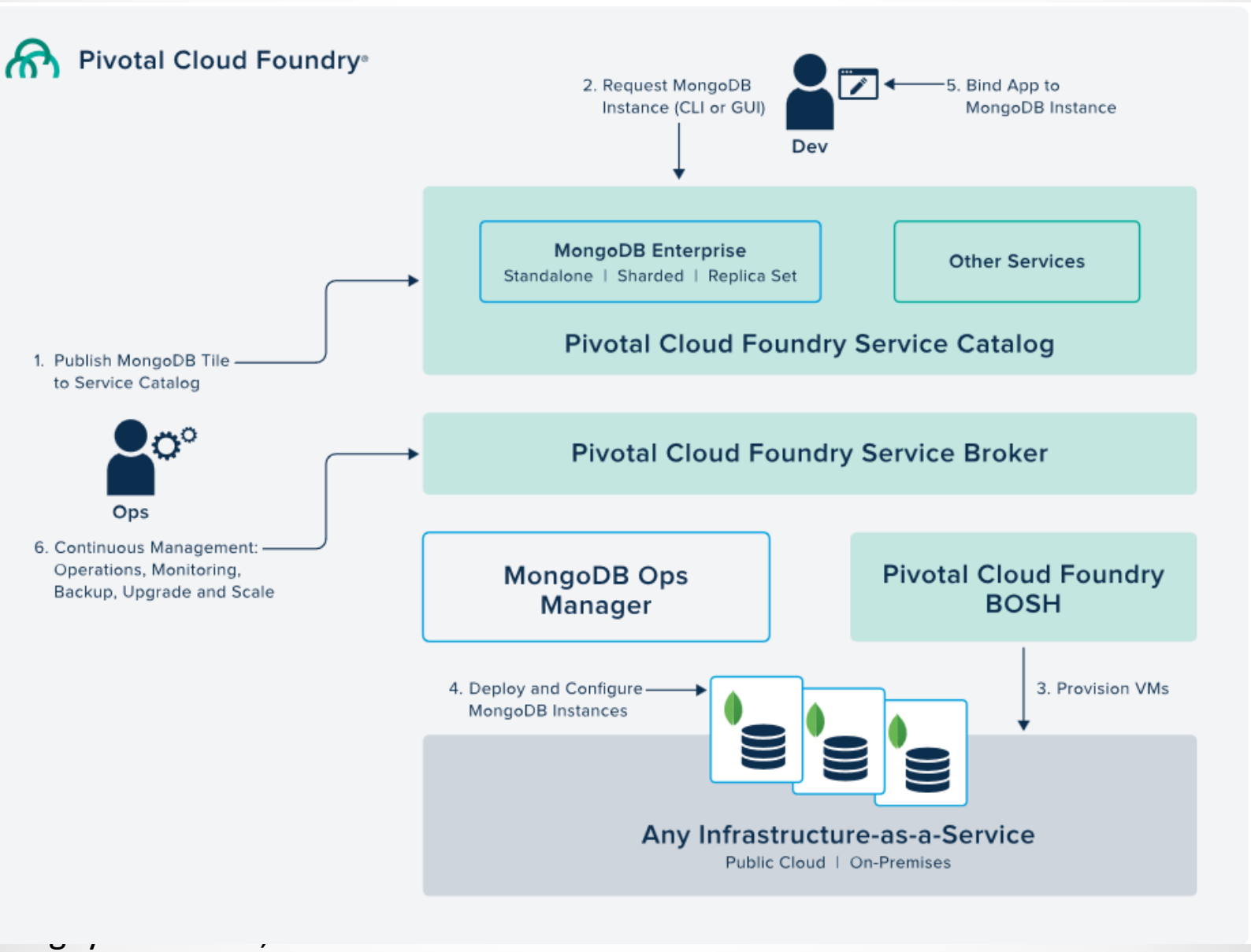



CLOUD NATIVE COMPUTING FOUNDATION

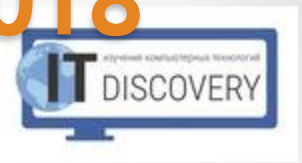
# Pivotal Cloud Foundry


- Spring Cloud Application platform (PAAS)
- Originally developed by VMWare
- Written in Ruby, Go and Java
- PCF can be deployed on AWS, VmWare vSphere, OpenStack, Google Cloud and Zure
- Provides services on-demand


Pivotal
Cloud Foundry
Pivotal
Application
Service
Pivotal
Container
Service
Pivotal
Function
Service
(Not yet Generally Available)
Pivotal
Services
Marketplace
Monitoring & Logging Services
Security Services
Networking Services
BOSH
Embedded OS
vSphere
OpenStack
AWS
Google
Cloud
Azure
Multi-Cloud


Pivotal Cloud Foundry®
2. Request MongoDB Instance (CLI or GUI)
Dev
5. Bind App to MongoDB Instance
MongoDB Enterprise
Standalone | Sharded | Replica Set
Other Services
Pivotal Cloud Foundry Service Catalog
1. Publish MongoDB Tile to Service Catalog
Ops
Pivotal Cloud Foundry Service Broker
6. Continuous Management: Operations, Monitoring, Backup, Upgrade and Scale
MongoDB Ops Manager
Pivotal Cloud Foundry BOSH
4. Deploy and Configure MongoDB Instances
3. Provision VMs
Any Infrastructure-as-a-Service
Public Cloud | On-Premises

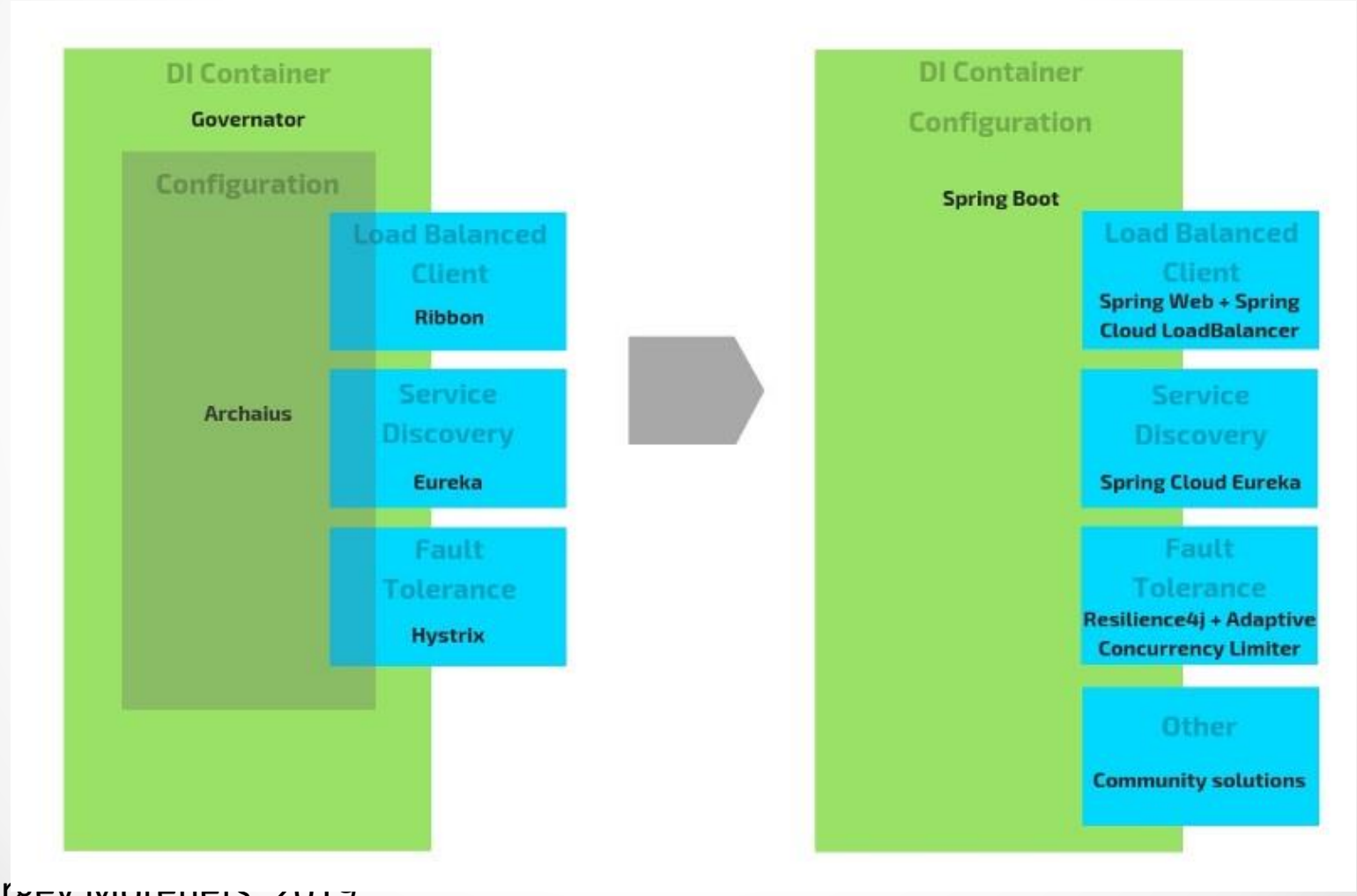
# Netflix & Spring Cloud.2014 - 2018




- ✓ Netflix stack created in 2007 and open-sourced in 2012

| Operations Component | Netflix, Spring, ELK |
|---|---|
| Service Discovery server | Netflix Eureka |
| Dynamic Routing and Load Balancer | Netflix Ribbon |
| Circuit Breaker | Netflix Hystrix |
| Monitoring | Netflix Hystrix dashboard and Turbine |
| Edge Server | Netflix Zuul |
| Central Configuration server | Spring Cloud Config Server |
| OAuth 2.0 protected API's | Spring Cloud + Spring Security OAuth2 |
| Centralised log analyses | Logstash, Elasticsearch, Kibana (ELK) |

# Netflix & Spring Cloud. 2019




- Sergey Morenets, 2019

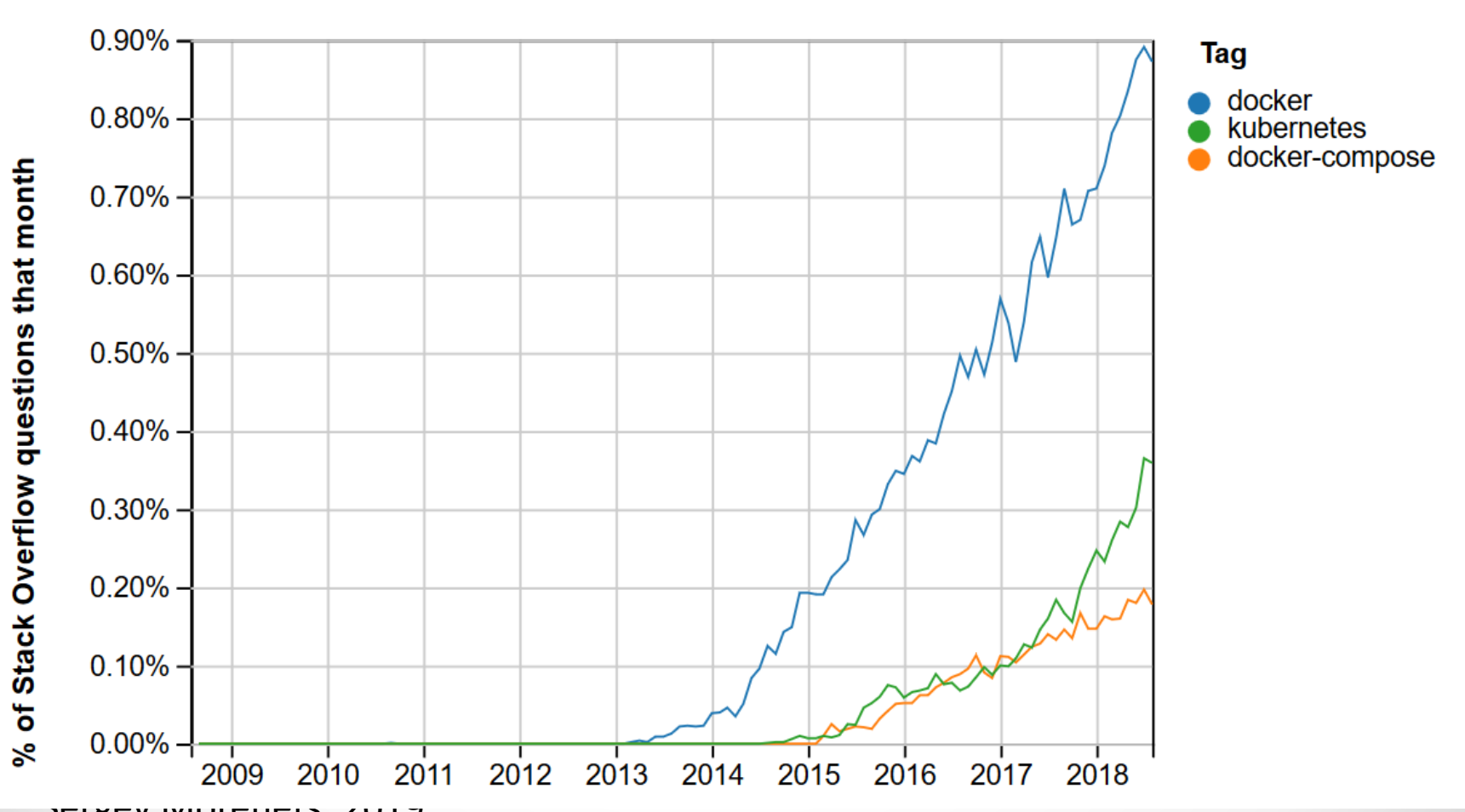
# Maintenance & replacement. 2019


| Stable technology | Replacement |
|---|---|
| Hystrix | Resilience4j |
| Hystrix Dashboard / Turbine | Spectator/Atlas/Micrometer + Monitoring System (Grafana, Graphite) |
| Ribbon | Spring Cloud Loadbalancer |
| Zuul 1.x | Spring Cloud Gateway |
| Archaius 1.x | Spring Boot external config + Spring Cloud Config |
| Governator | Spring Framework |

# Spring Cloud 2.x


- Released in June 2018
- Requires Java 8
- Added Spring Cloud Function/Spring Cloud Gateway/Spring Cloud Loadbalancer
- New starters
- Spring Cloud OpenFeign project
- Spring Cloud Aws/CloudFoundry projects enhancements
- Current version 2.1.2

# Microservices deployment




Sergey Morenets, 2019

# Application configuration


- ✓ Resource management
- ✓ Resource settings(DB, network) storage

# Configuration options

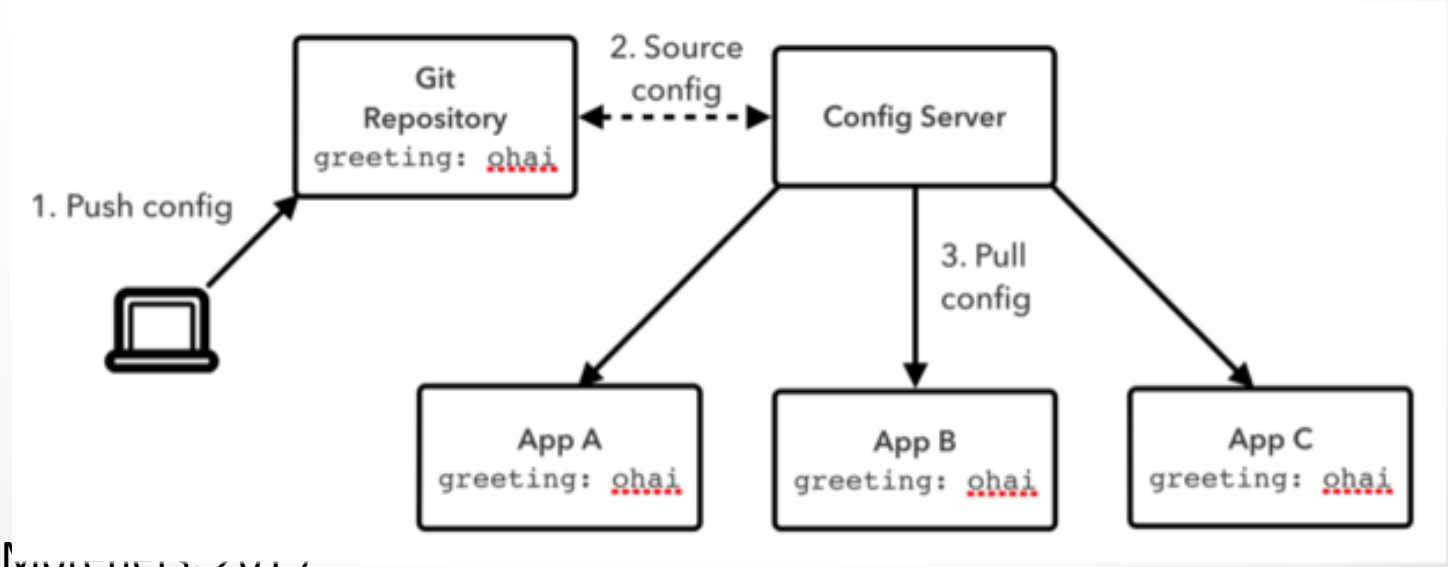

- ✓ Application-specific(requires restart)
- ✓ File system(unsupported for cloud)
- ✓ Environment variables
- ✓ Cloud-specific

# Solution


- ✓ Cloud-independent
- ✓ Centralized
- ✓ Doesn’t require app restart
- ✓ Self-management

# Spring Cloud Config

- ✓ Standalone server that stores configuration settings
- ✓ Clients communicate with server via HTTP and obtain global configuration settings



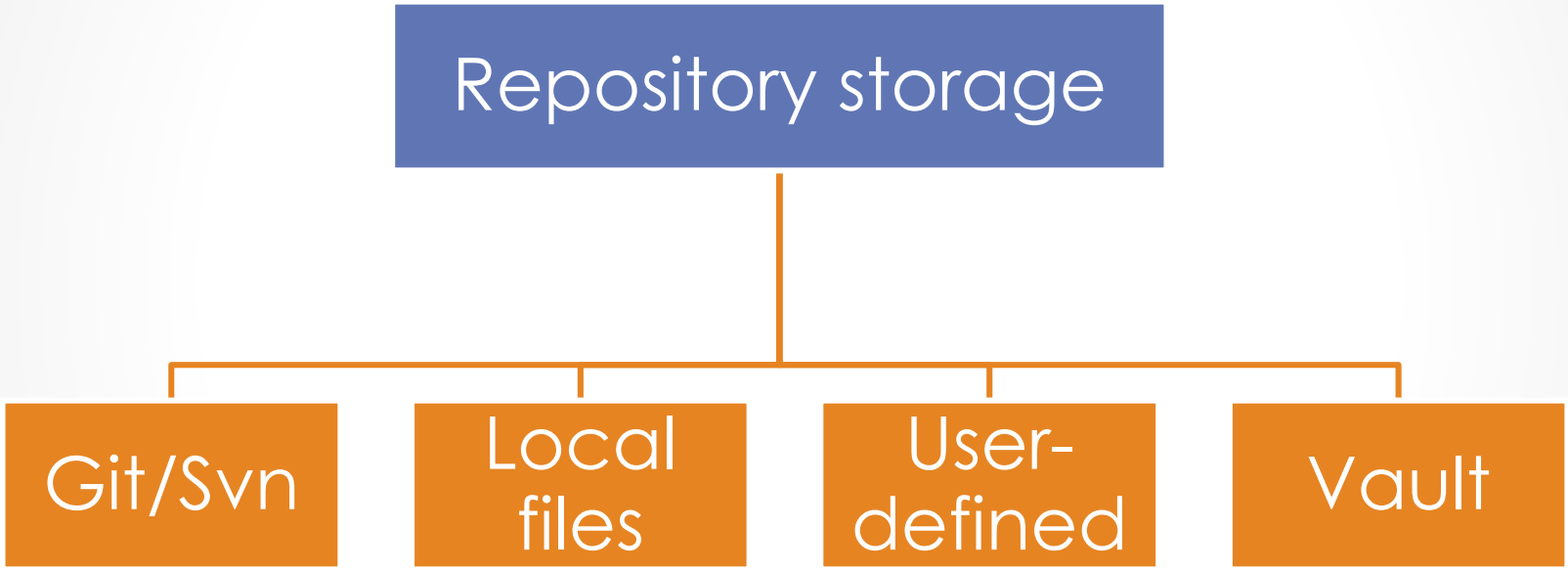
# Spring Cloud Config


```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-parent</artifactId>
            <version>${spring.cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

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

# Spring Cloud Config


```
@SpringBootApplication
@EnableConfigServer
public class MainApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class, args);
    }
}
```


Repository storage
Git/Svn
Local files
User-defined
Vault

# Config server. Git configuration


```
spring.cloud.config.server.git.uri=https://github.com/it-discovery/microservices
spring.cloud.config.server.git.search-paths=config
```

application.properties

Path to GitHub repository

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/it-discovery/microservices
          searchPaths: config
```

application.yml

Folder in GitHub repository

# YAML


```yaml
---
key: value
map:
    key1: "foo:bar"
    key2: value2
list:
  - element1
  - element2
# This is a comment
listOfMaps:
  - key1: value1a
    key2: value1b
  - key1: value2a
    key2: value2b
---
```

```
some
other
listing
```

# Task 3. Configuration server

1. Import **microservices** project into your IDE
2. Update ConfigServerApplication class
3. Update **application.properties** file.
4. Make executable jar file using command: **mvn install spring-boot:repackage** and run it as ordinary java file.

# Configuration selection


- application.properties/application.yml files store global configuration for all clients
- {application} ← **spring.application.name**
- {profile} ← **spring.profiles.active**
- {label} ← **spring.cloud.config.label**

# Client configuration


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

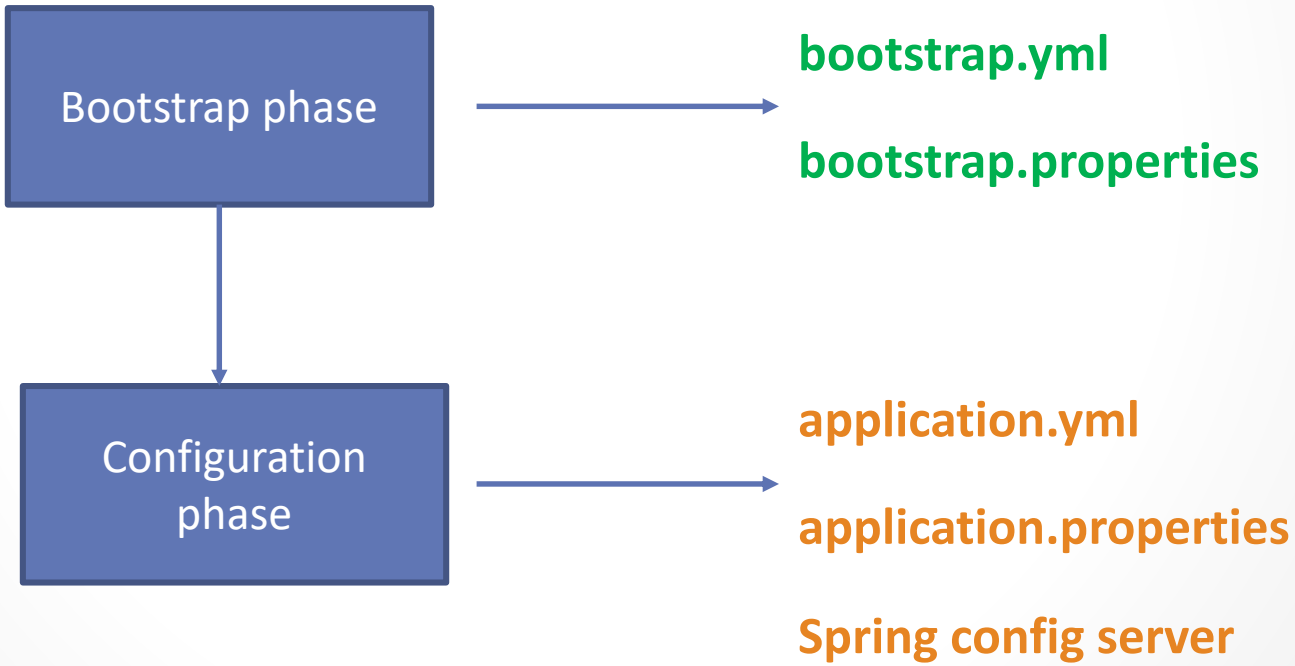
```
spring:
  application:
    name: library-client
  cloud:
    config:
      uri: http://localhost:8000
```

Client service identifier

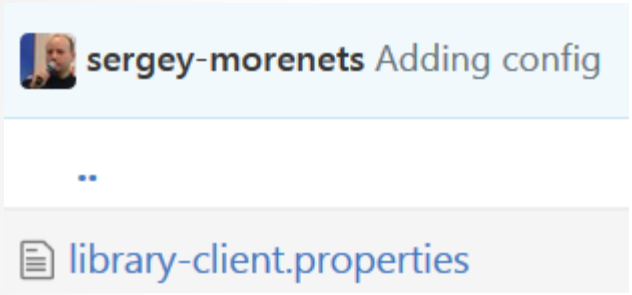
bootstrap.yml

# Spring configuration phases


**Load information about configuration source**



# GitHub repository


```
company1 = Crunchify
company2 = Google
Crunchify_Address = NYC,US
Google_Address = Mountain View,CA,US
```

# Task 4. Client configuration

1. Add new dependency to the book application:
2. Add **bootstrap.properties** to the book application, populate application name (**application**) and cloud configuration URI.
3. Populate **libraryName** field in BookController from Spring configuration (it should be taken from GitHub repository).
4. Make executable jar file using command: **mvn install spring-boot:repackage** and run it as ordinary java file.

# Local files. Config server


```
#spring.cloud.config.server.git.uri=https://github.com/
#spring.cloud.config.server.git.search-paths=config

spring.profiles.active=native
```

application.properties

```
company1 = Crunchify
company2 = Google
Crunchify_Address = NYC,US
Google_Address = Mountain View,CA,US
```

{spring.application.name}.properties

# Client configuration. Spring profiles


```yaml
spring:
  application:
    name: library-client
  cloud:
    config:
      uri: http://localhost:8000
  profiles:
    active:
    - dev
```

bootstrap.properties

```
company1 = Crunchify
company2 = Google
Crunchify_Address = NYC,US
Google_Address = Mountain View,CA,US
```

{spring.application.name}-dev.properties

# Overriding properties


- By default configuration in application.properties/application.yml is shared between all applications
- Local configuration cannot override by default remote configuration
- You can also setup global immutable configuration

```
spring:
  cloud:
    config:
      server:
        overrides:
          client-name: Google
```

Config server. application.yml

# Security


- ✓ Encryption and decryption are supported
- ✓ Server-side and client-side encryption are available
- ✓ Requires **Java Cryptography** extension installed (server-side)
- ✓ Manual encryption using /encrypt & decrypt endpoints
- ✓ Config server can use symmetric or asymmetric keys

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

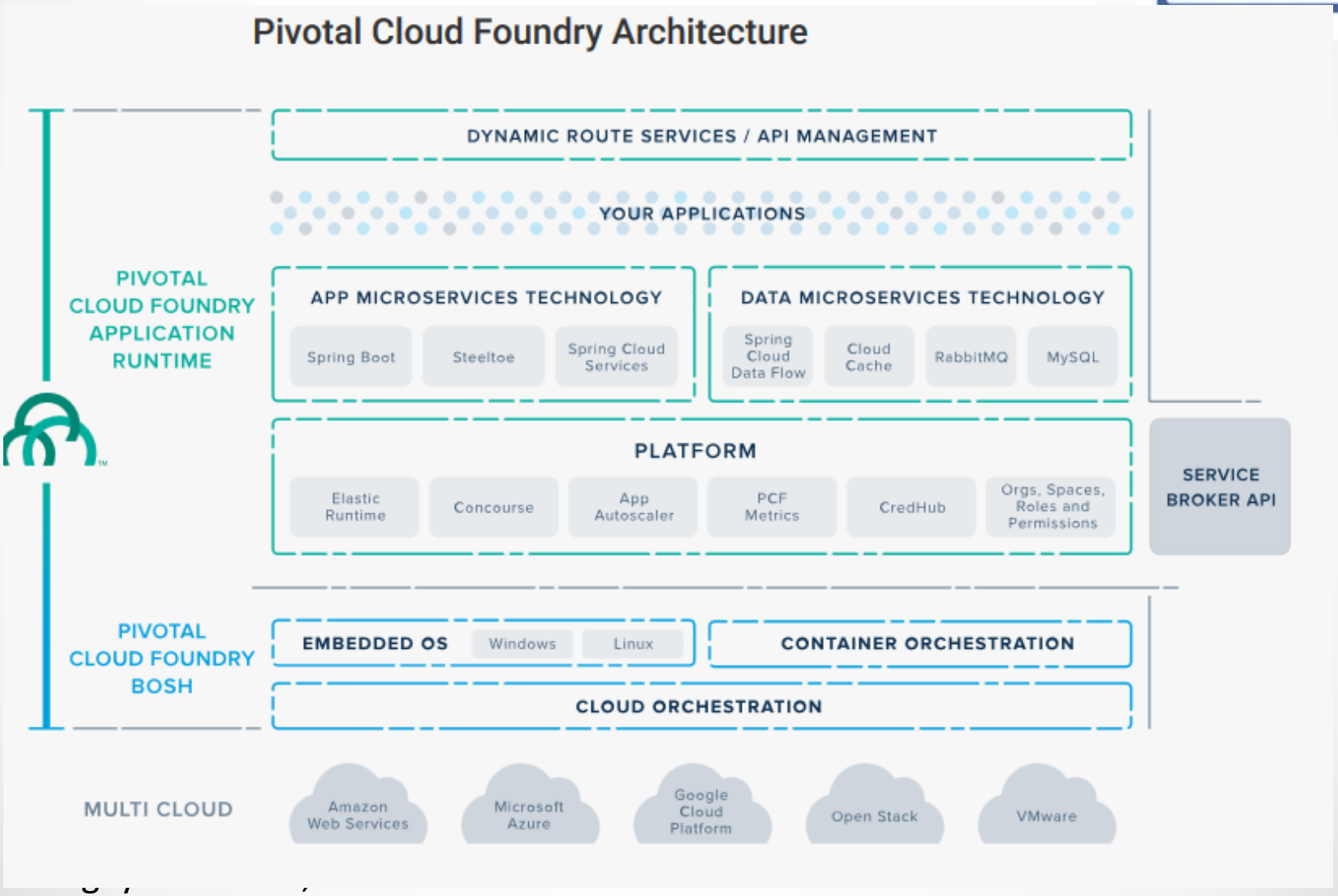
application.yml

```
spring.cloud.config.server.encrypt.enabled=true
```

# Task 5. Local configuration

1. Update **application.properties** in the config-server application to switch to local files storage.
2. Copy properties file from GitHub repository to the **src/main/properties** folder and rename it to **library-client.properties**. Rename book application name:
3. Restart **config-server** and book application
4. Copy properties file into new file with "**dev**" suffix. Update **bootstrap.properties** in the client application and set **Spring** profile to "dev"

# Deployment


Pivotal Cloud Foundry Architecture
DYNAMIC ROUTE SERVICES / API MANAGEMENT
YOUR APPLICATIONS
PIVOTAL CLOUD FOUNDRY APPLICATION RUNTIME
APP MICROSERVICES TECHNOLOGY
Spring Boot
Steeltoe
Spring Cloud Services
DATA MICROSERVICES TECHNOLOGY
Spring Cloud Data Flow
Cloud Cache
RabbitMQ
MySQL
PLATFORM
Elastic Runtime
Concourse
App Autoscaler
PCF Metrics
CredHub
Orgs, Spaces, Roles and Permissions
SERVICE BROKER API
PIVOTAL CLOUD FOUNDRY BOSH
EMBEDDED OS
Windows
Linux
CONTAINER ORCHESTRATION
CLOUD ORCHESTRATION
MULTI CLOUD
Amazon Web Services
Microsoft Azure
Google Cloud Platform
Open Stack
VMware

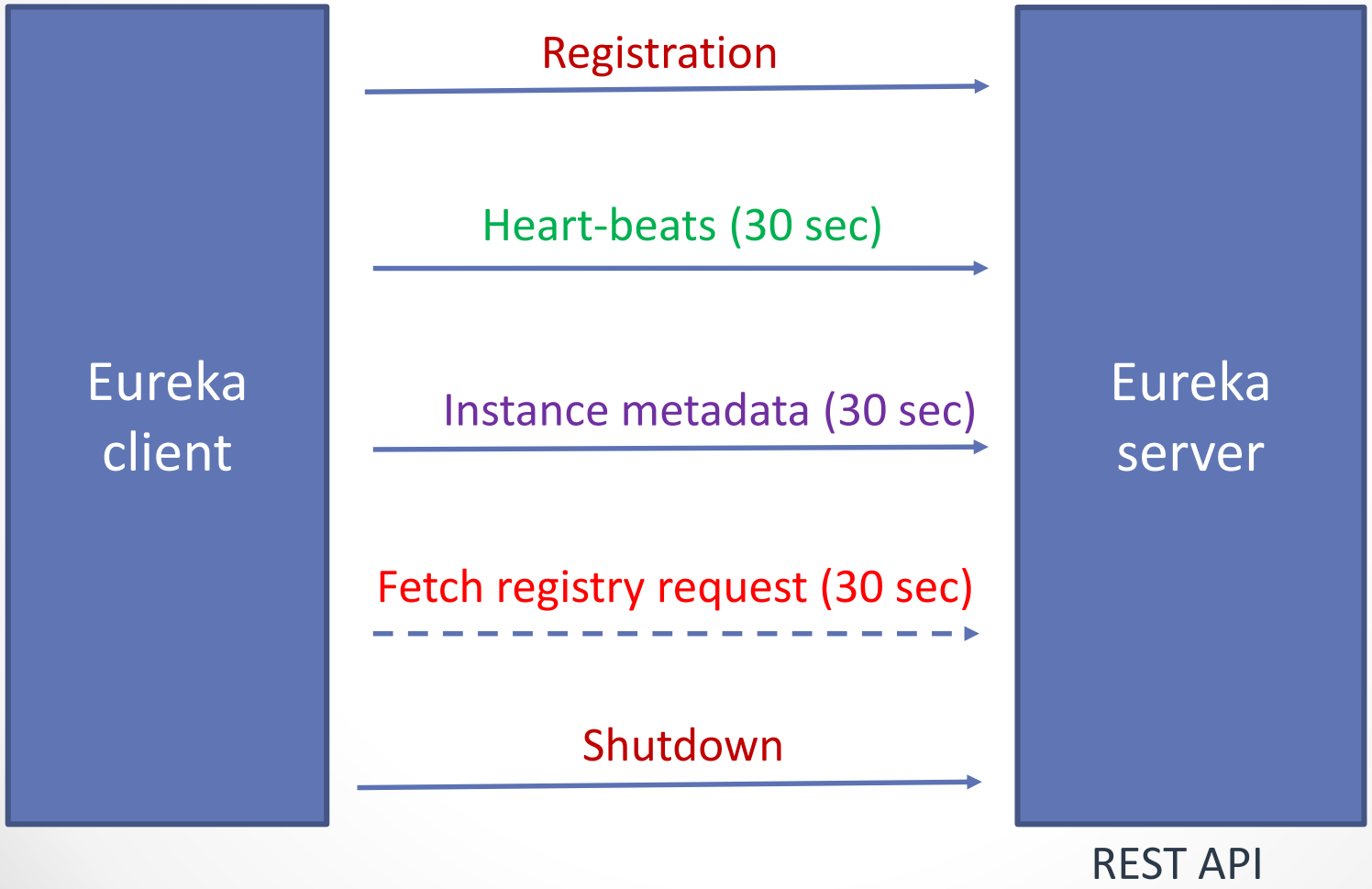
# Service discovery

- Client should register itself and obtain information about other clients
- Single lookup service
- Solutions: Eureka, Consul, Zookeeper

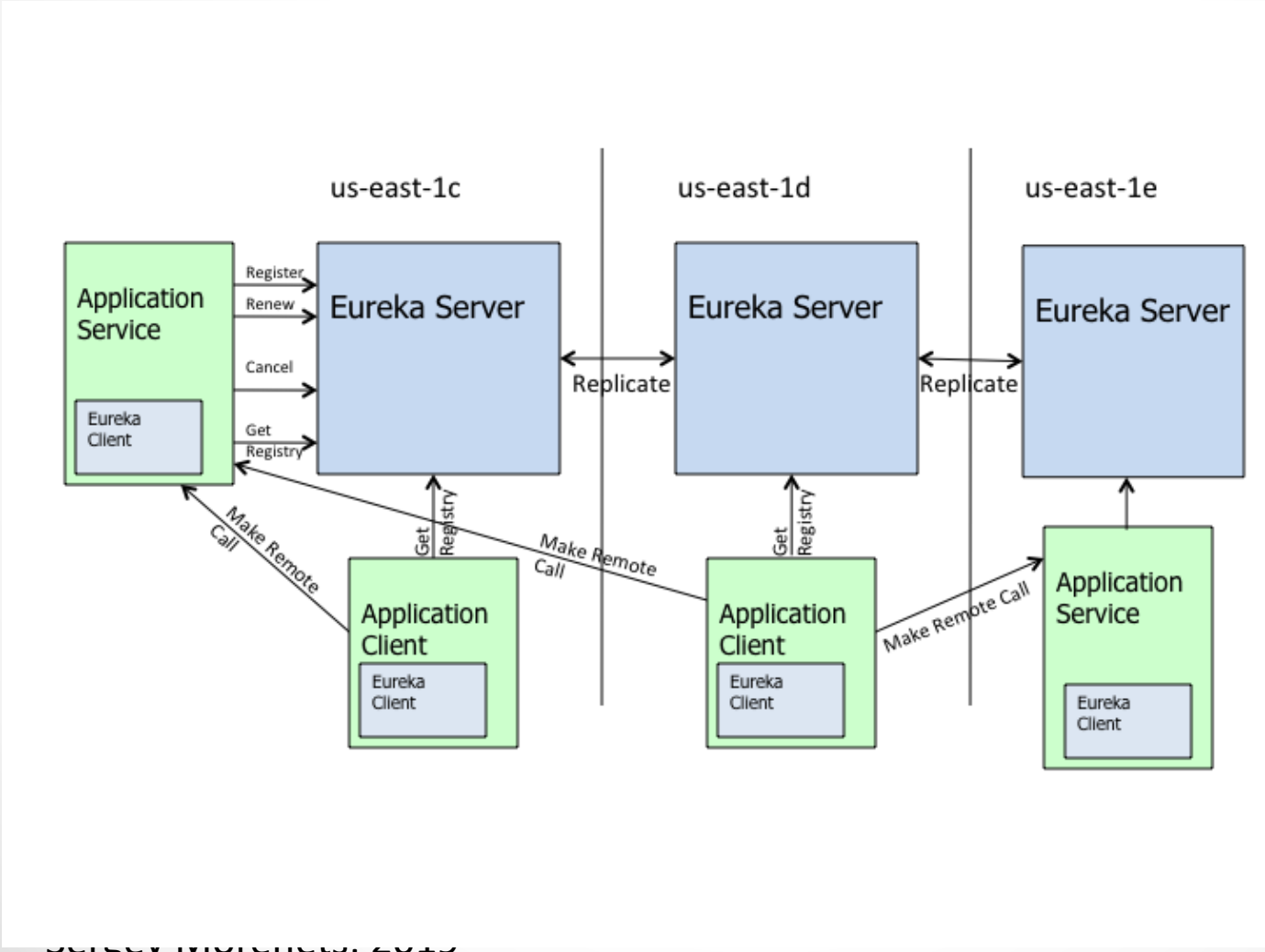
# Eureka


- Service discovery client/server
- Has two component: **Eureka Client** and **Eureka Server**
- **Application client** uses Eureka Client to communicate with Eureka Server
- Eureka Client uses **Jersey** client for HTTP communication (can be changed to RestTemplate)
- High availability achieved by running multiple servers in different zones/regions sharing their state
- Client registrations are stored in memory
- Designed to run mostly in AWS (Amazon EC2)
- Requires at least one peer

# Eureka. Message flow



# Eureka availability zones. AWS



# Eureka regions&availability zones


Region

Availability zones

```
eureka.us-east-1.availabilityZones=us-east-1c,us-east-1d,us-east-1e

eureka.serviceUrl.us-east-1c=http://ec2-5.amazonaws.com:7001/eureka/v2/
eureka.serviceUrl.us-east-1d=http://ec2-2.amazonaws.com:7001/eureka/v2/
eureka.serviceUrl.us-east-1e=http://ec2-0.amazonaws.com:7001/eureka/v2/
```

Eureka server. eureka-client.properties

# Eureka server configuration


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

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class,
                args);
    }
}
```

# Eureka server configuration


```
server.port=8090
eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

application.properties

# Eureka server configuration


| Setting | Description | Default |
|---|---|---|
| eureka.client.eurekaServiceUrlPollIntervalSeconds | Time to check for changes in Eureka servers | 5 min |
| eureka.instance.leaseRenewalIntervalInSeconds | Interval time to send heartbeats (from client to server) | 30 sec |
| eureka.instance.leaseExpirationDurationInSeconds | Time to remove client from instance after not receiving heartbeat | 90 sec |
| eureka.client.instanceInfoReplicationIntervalSeconds | Interval time to send instance information from client to server | 30 sec |
| eureka.client.registryFetchIntervalSeconds | Interval time to query Eureka server registry | 30 sec |

# Eureka server REST API


| Endpoint | Description |
|---|---|
| /eureka/apps | Returns all the registered instances |
| /eureka/apps/<app.id> | Returns information about all <application.name> instances |
| /eureka/apps/<app.id>/<instanceId> | Returns information about specific instance |
| /eureka/instances/<instanceId> | Returns information about specific instance |
| PUT /eureka/apps/<app.id>/<instanceId> | Send client heartbeat |
| POST /eureka/apps/<app.id> | Register new application instance |

# Eureka client configuration


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

```
@EnableDiscoveryClient
@SpringBootApplication
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class,
                args);
    }
}
```

```
eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka/
```

application.properties

# Eureka client configuration


```
eureka:
  client:
    registerWithEureka: true
    fetchRegistry: true
    useDnsForFetchingServiceUrls: true
    eureka-server-d-n-s-name: eureka.local
    eurekaServerPort: 8761
    eureka-server-u-r-l-context: eureka
    region: us-east-1
    availabilityZones:
      us-east-1: us-east-1a,us-east-1b
    preferSameZoneEureka: true
```

# Eureka client. Manual discovery

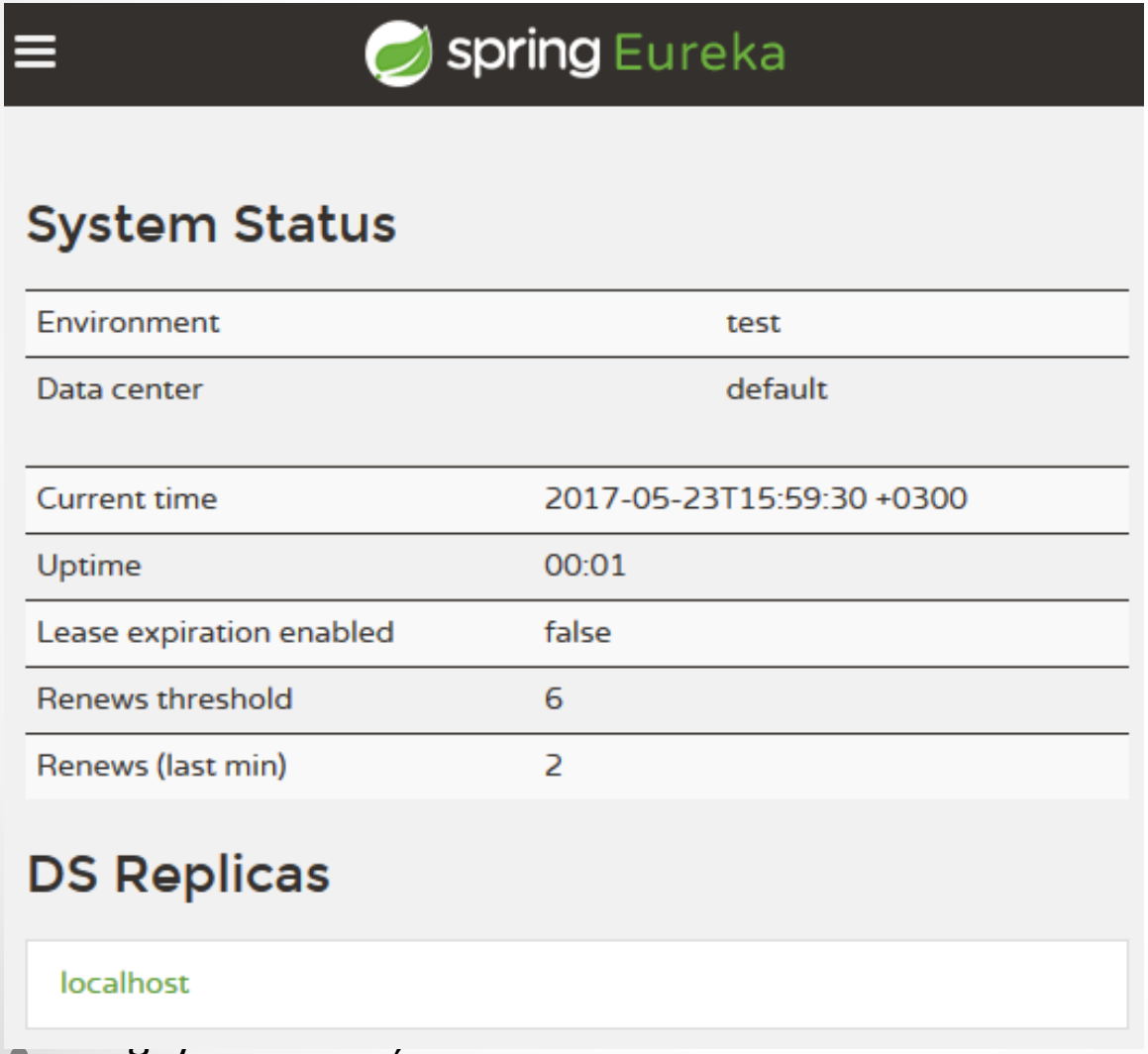

```
@Autowired
private DiscoveryClient discoveryClient;

public List<ServiceInstance> getInstances(
        String serviceId) {
    return discoveryClient.getInstances(serviceId);
}
```

```
spring.application.name=library-client
```

# Eureka web UI


http://localhost:8090/



# Eureka web UI


http://localhost:8090/

## Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| CONFIG-SERVICE | n/a (1) | (1) | UP (1) - SERGEY:config-service:8000 |
| LIBRARY-CLIENT | n/a (1) | (1) | UP (1) - SERGEY:library-client:8003 |
| LIBRARY-LOGGING | n/a (1) | (1) | UP (1) - SERGEY:library-logging |

# Resilience to failures


EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP
THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

| Status |
| --- |
| UP (1) - SERGEY:config-service:8000 |
| UP (2) - SERGEY:library-client:8085 , SERGEY:library-client:8003 |
| UP (2) - SERGEY:library-logging , SERGEY:library-logging:8085 |

```
eureka:
  server:
    enableSelfPreservation: false
```

Eureka server. application.yml

At least two Eureka instances must be running

# Task 6. Eureka server

1. Add @EnableEurekaServer annotation to the EurekaApplication class.
2. Populate **application.properties** file for eureka project.
3. Populate **bootstrap.properties** with eureka address.
4. Start client project and observe in the eureka/client logs that client registers itself on the Eureka server.

# Eureka/Config Server integration


```
spring.application.name=config-service
```

Configuration server. application.properties

```
spring:
  cloud:
    config:
      discovery:
        service-id: config-service
        enabled: true
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8090/eureka/
```

Client application. bootstrap.yml

# Config server retry pattern


```xml
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
```

Client application. pom.xml

```yaml
spring:
  cloud:
    config:
      fail-fast: true
      retry:
        max-attempts: 5
        initial-interval: 1000
        max-interval: 5000
        multiplier: 2
```

Client application. application.properties

Exponential back-off strategy

# Task 7. Eureka & Config server

1. Add **eureka-server** dependency for the config-server project.
2. Add **@EnableDiscoveryClient** annotation for the **ConfigServerApplication** class.
3. Add properties for config-server project.
4. Update **bootstrap.properties** of the client applications to add new properties:
5. Start **Eureka/config-server/client applications** and make they work properly together.

# Eureka server. Peer awarness


- Every server know about other peers and replicate the service registry with them to support resilience and load balancing
- Client connects to first server in the list and send heartbeats
- If the first server becomes unavailable all clients migrate to the next server
- If the first server becomes available all clients migrate back to the first server
- If neighboring peer become offline the peer goes into **preservation mode** until nearby peer gets online

# Eureka server. Peer awarness


```yaml
spring:
  profiles: peer1
server:
  port: 8090
eureka:
  instance:
    hostname: peer1
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://peer1:8090/eureka/,http://peer2:8091/eureka/
```

application.yml

# Task 8. Eureka peer-to-peer

1. Create new Docker service eureka-server2 (based on **eureka-server** project) and specify another port for it (for example, 8091). You can also use Docker Swarm/Kubernetes to run multiple instances of Eureka
2. Update eureka-server project server configuration (**defaultZone**) so that it supports multiple peers.
3. Run applications and  open Web UI http://localhost:8090/ and observer registered clients


✓ Sergey Morenets, sergey.morenets@gmail.com