

Agenda

- **Introduction to microservice**
- **Spring cloud and related technologies**
- **Examples and applications**
- **Netflix tools: Eureka, Ribbon, Hystrix**
- **Example application**

Monolith vs



Microservices



Classical monolith design

Web Layer

(controllers, exception handlers, filters, view templates, and so on)

Service Layer

(application services and infrastructure services)

Repository Layer

(repository interfaces and their implementations)



What is microservice?

“Microservice architectural style is an approach to developing a single application as a suite of **small services**, each running in its **own process** and communicating with **lightweight** mechanisms, often an HTTP resource API.”

Martin Fowler, ThoughtWorks

What is microservice? Another defination

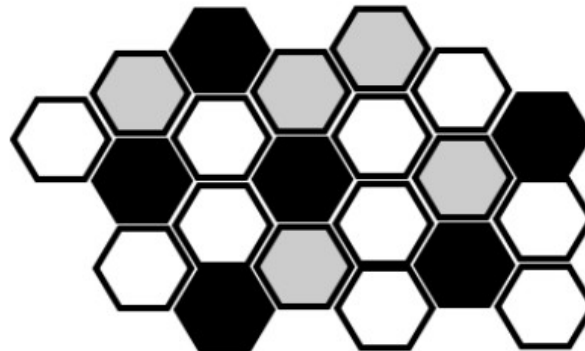
What are microservices?

Microservices are an architecture style used by many organizations today as a game changer to achieve a high degree of agility, speed of delivery, and scale. Microservices give us a way to develop more physically separated modular applications.

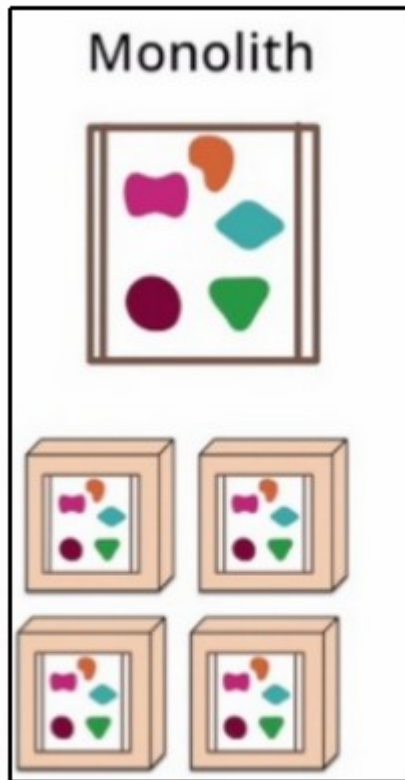
Microservices are not invented. Many organizations such as Netflix, Amazon, and eBay successfully used the divide-and-conquer technique to functionally partition their monolithic applications into smaller atomic units, each performing a single function. These organizations solved a number of prevailing issues they were experiencing with their monolithic applications.

Microservices – the honeycomb analogy

The honeycomb is an ideal analogy for representing the evolutionary microservices architecture.



Monolithic architecture



Classical architecture.

Application is built as **a single unit.**

Typical 3 layer EA:

- client-side UI (Browser, HTML + JS)
- a database (RDBMS, NoSql ..)
- server-side application

(Java, .NET, Ruby, Python, PHP ..)

Monolithic Attributes

The server-side application

- will handle HTTP requests
- execute domain logic
- retrieve and update data from the database
- select and populate HTML views

This server-side application is a

Monolith - Single logical executable.

Monolithic server - **natural approach**

All logic for handling a request runs in a single process, **divided** and **organized** into classes, functions, and namespaces.

Application is developed on a developer's laptop, deployed to a testing environment and after that to production environment.

Monolith is horizontally scaled by running many instances behind a **load-balancer**.

Monolithic Why not?

Any changes to the system involve building and deploying a new version of the application.

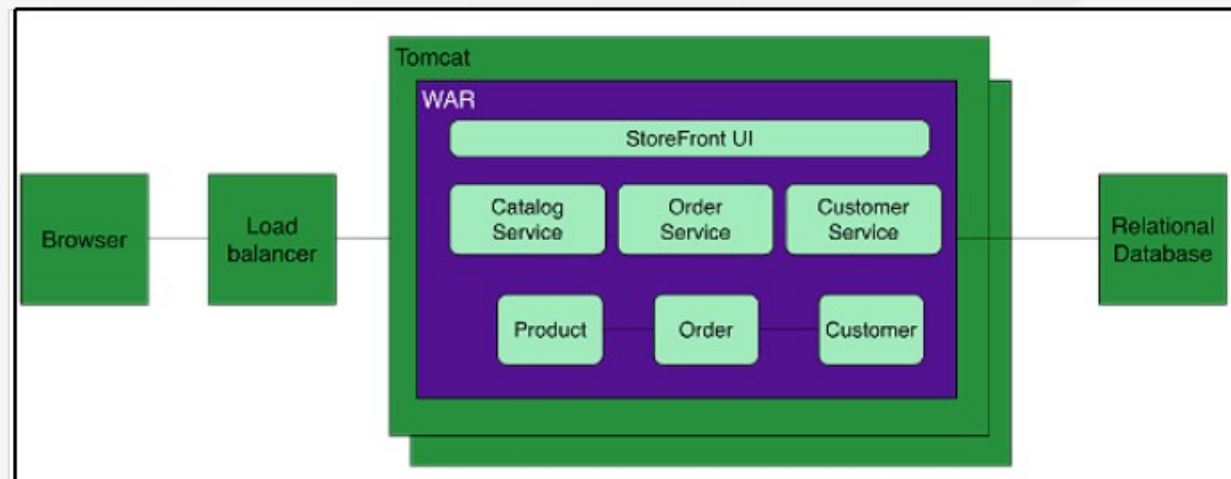
Changes require good **planning** and **coordination**.

Changes are **expensive**.

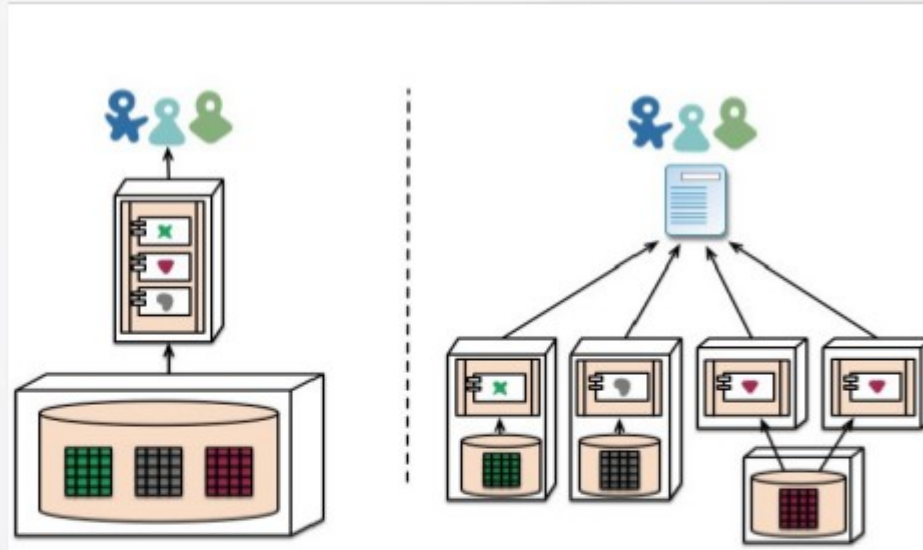
It is hard to keep a good **modular structure**.

Scaling requires scaling of the entire application rather than parts of it that require greater resource.

Long release cycles.



Microservice attributes



Decentralized data management.

Decentralized governance.

Microservice attributes

They are built around business capabilities.

Each component runs as a **separate application**, clustered to as many nodes as required.



Microservice vs SOA

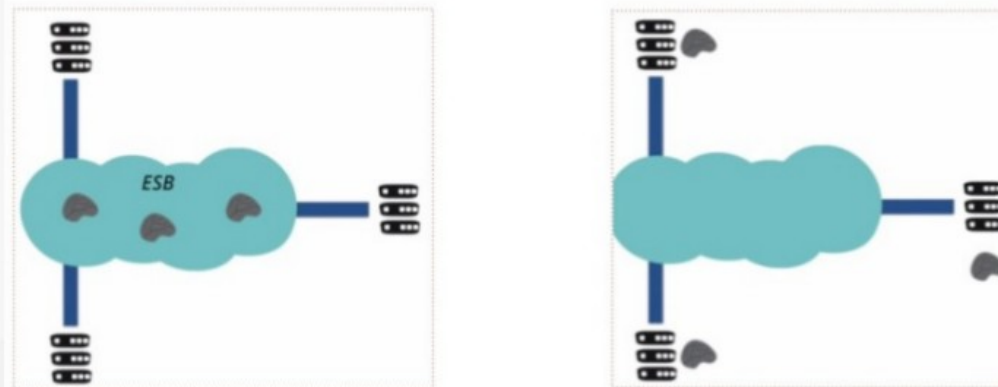
Applications aim to be as **decoupled** and as **cohesive** as possible.

Components usually communicate through **REST**.

Components sometime use **lightweight messaging** (RabbitMQ, Zero MQ).

There is **NO ESB** or any other form of central communication management.

Smart endpoints an dumb pipes:



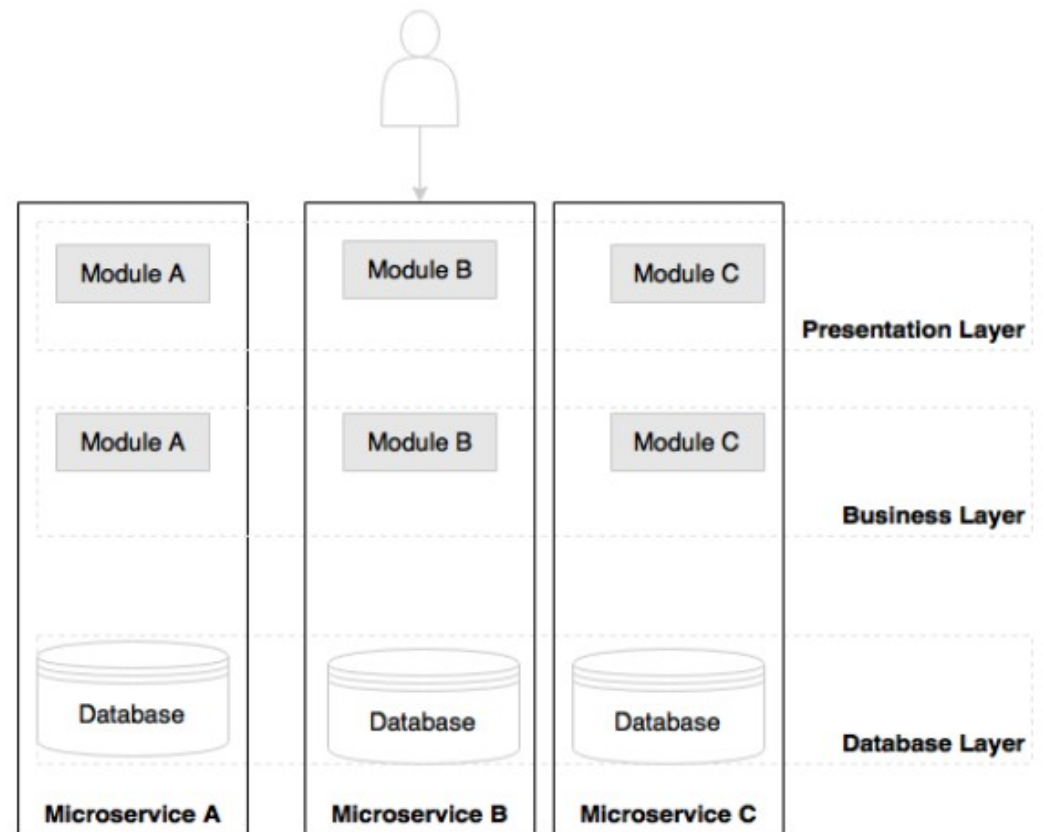
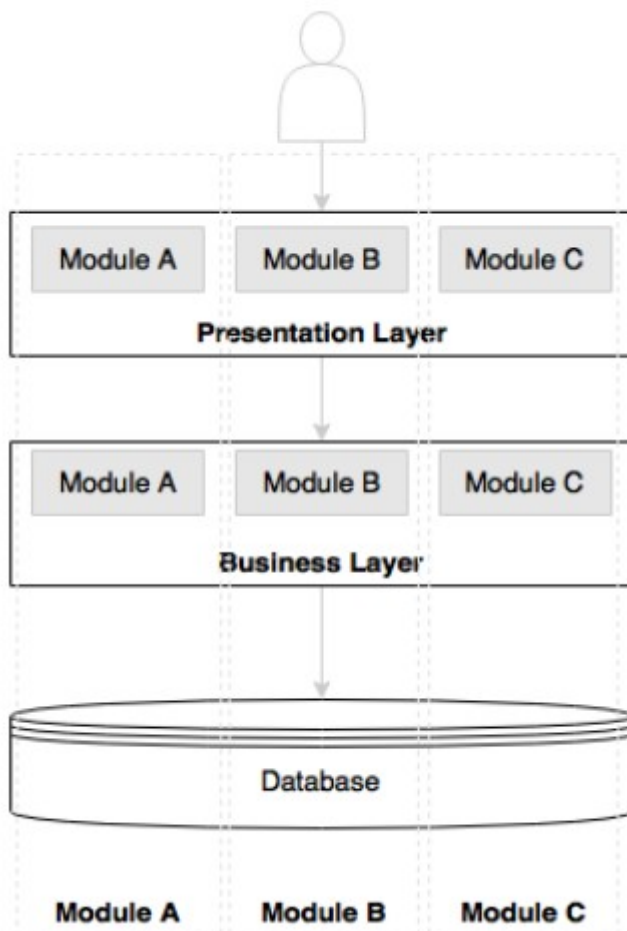
Microservice vs SOA

SOA	Microservices
XML	JSON
Complex to integrate	Easy to integrate
Heavy	Lightweight
Requires tooling	Light tooling
HTTP/SOAP	HTTP/REST

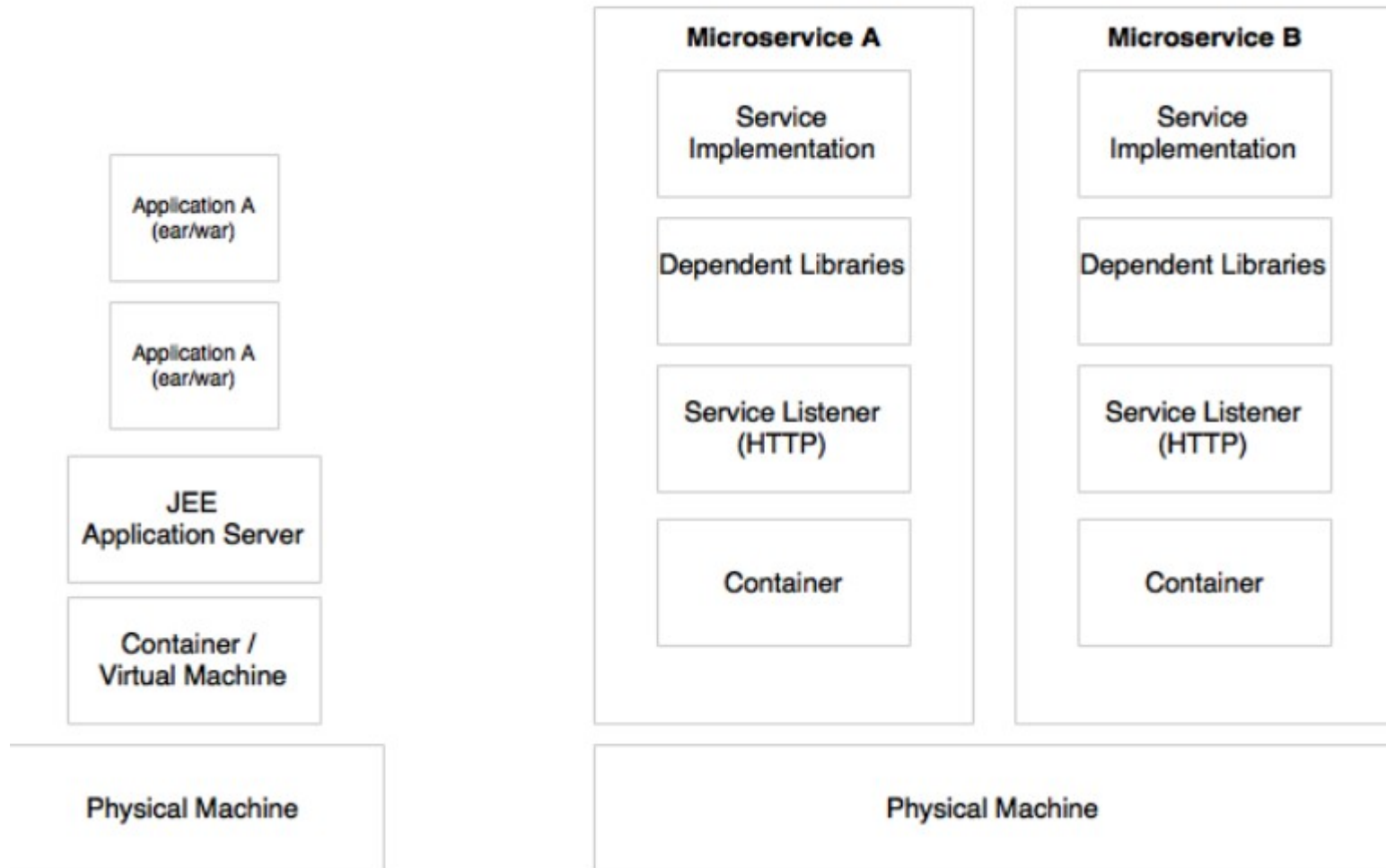
Advantage of microservice

- **INDEPENDENT SCALING**
 - Each microservice can scale independently
- **INDEPENDENT UPGRADES**
 - Each microservice can deployed independently
 - Teams are independent
- **EASY MAINTENANCE**
 - Each microservice does one feature : code more readable
- **POTENTIAL HETEROGENEITY AND POLYGLOTISM**
 - Teams may have # platform, language
- **FAILURE AND RESOURCE ISOLATION**
 - If one service crash, it will not affect the rest of application
- **IMPROVED COMMUNICATION ACROSS TEAMS**
 - Membres of domain work in single team

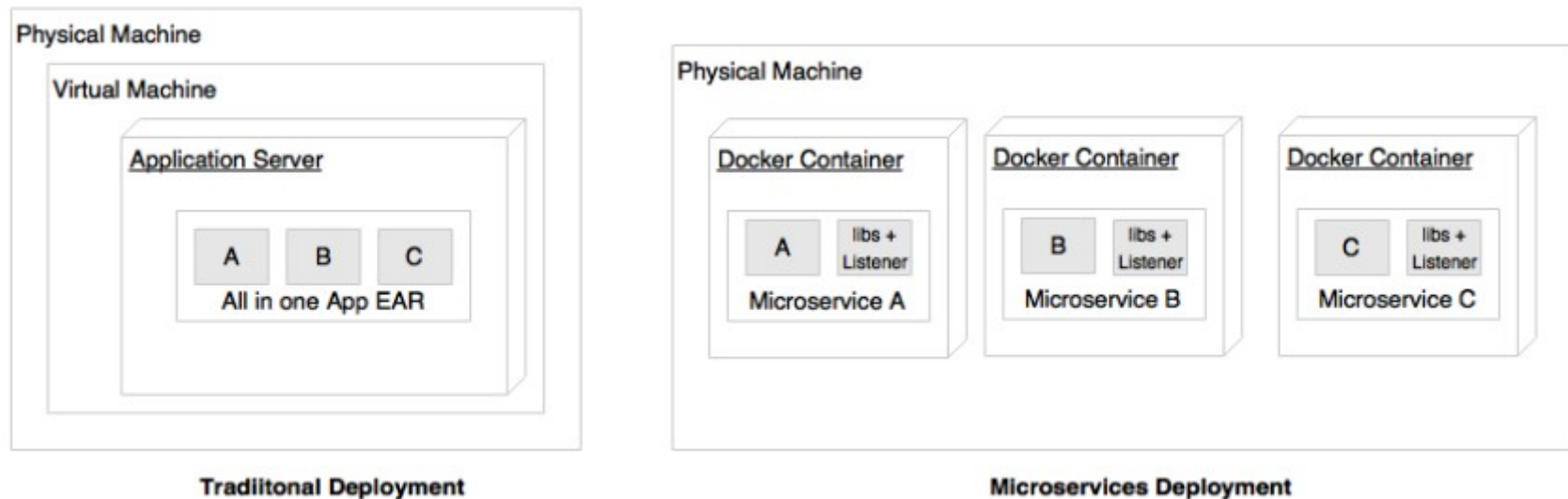
Monolith vs Microservice



Microservices are autonomous



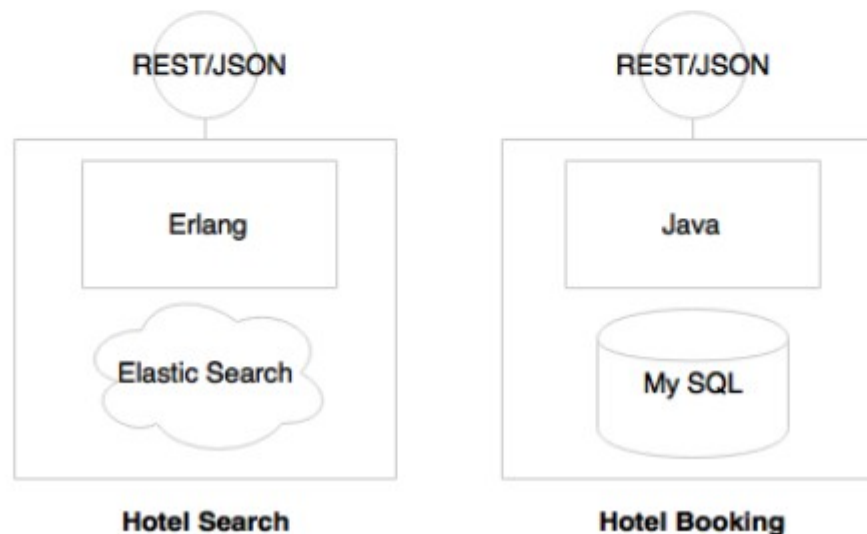
Deployment : monolith vs ms



As shown in the preceding diagram, microservices are typically deployed in Docker containers, which encapsulate the business logic and needed libraries. This helps us quickly replicate the entire setup on a new machine or on a completely different hosting environment or even to move across different cloud providers. As there is no physical infrastructure dependency, containerized microservices are easily portable.

Microservices with polyglot architecture

- 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.



Automation in a microservices environment

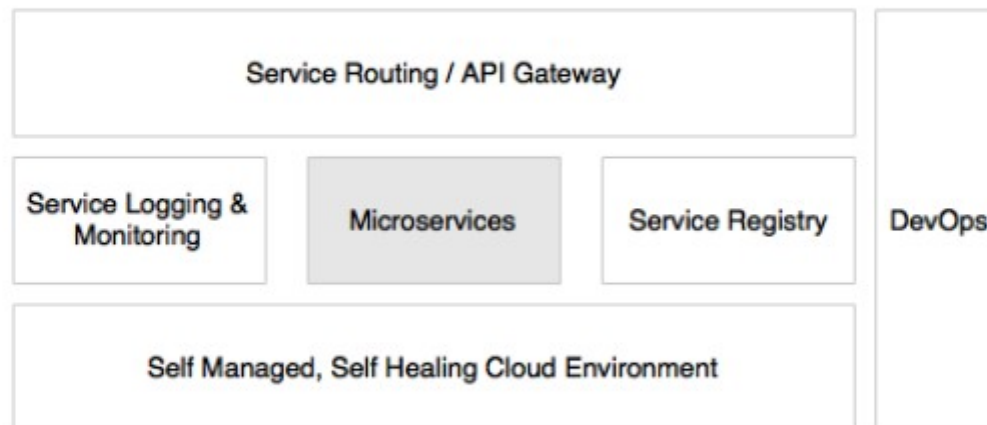
Most of the microservices implementations are automated to a maximum from development to production.

As microservices break monolithic applications into a number of smaller services, large enterprises may see a proliferation of microservices. A large number of microservices is hard to manage until and unless automation is in place. The smaller footprint of microservices also helps us automate the microservices development to the deployment life cycle. In general, microservices are automated end to end – for example, automated builds, automated testing, automated deployment, and elastic scaling.



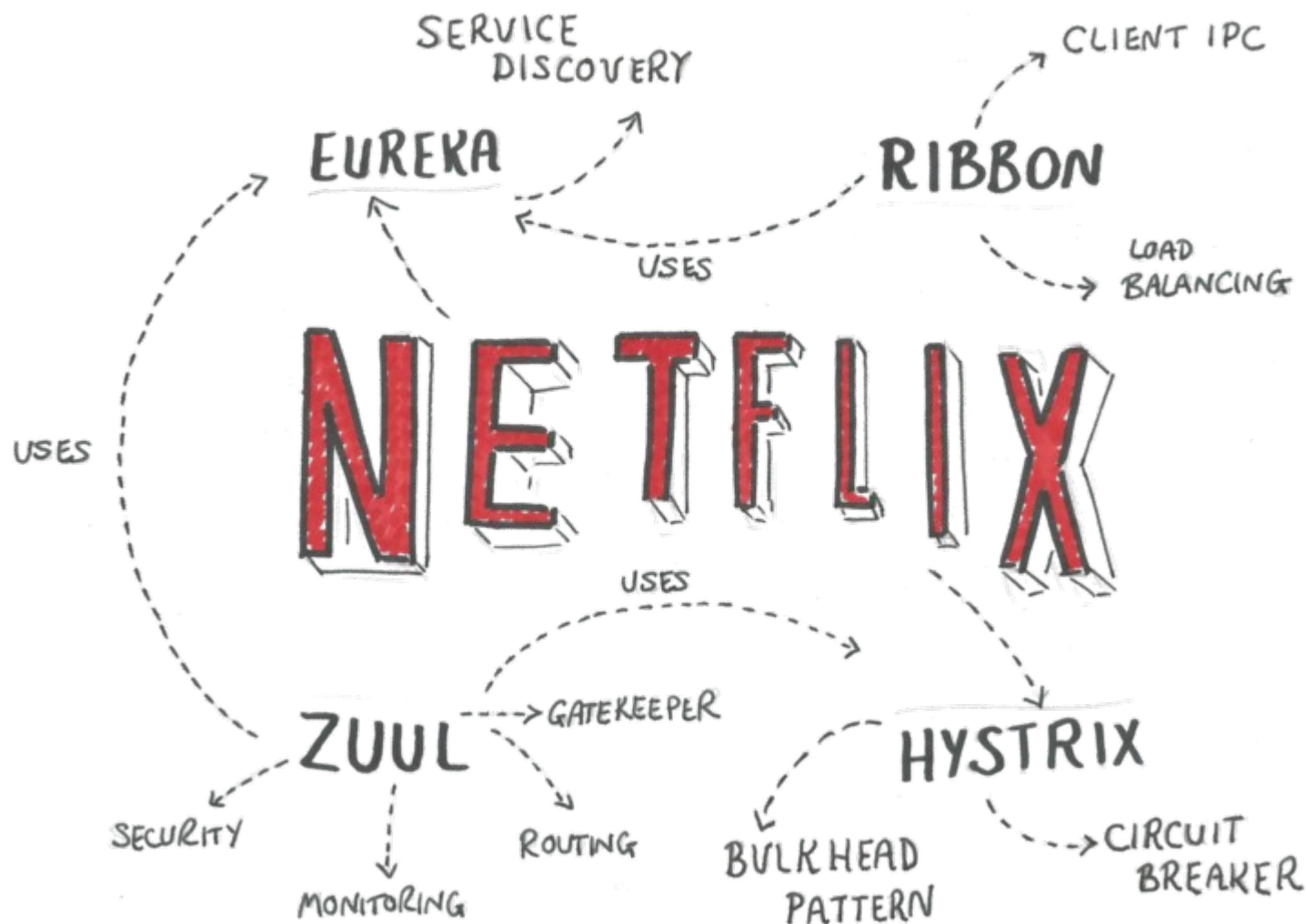
Microservices with a supporting ecosystem

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, and flow control mechanisms.



Microservices work well when supporting capabilities are in place, as represented in the preceding diagram.

Introducing Netflix and their cloud computing technology



Why Spring boot?

- Getting Started w/ Spring easy
- Convention over Configuration
- SPRING CLI
- Entry Point
- Starter POMs
- Production Ready
- Run anywhere!

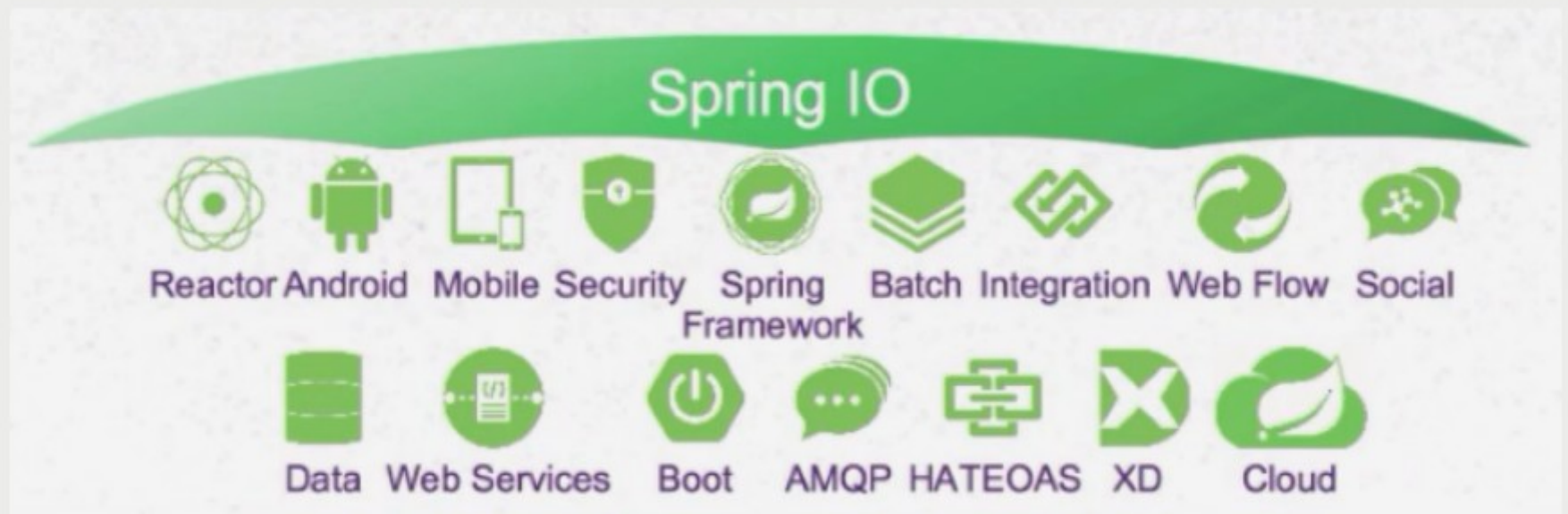


Spring boot best for creating microservice

Introduction to Spring cloud

1- Spring Cloud Overview

Spring is a platform built for developing web applications in the **Java** language. It was first introduced in 2004. In 2006, sub-projects appeared. Each sub-project focuses on a different field. Till now, you can see the sub-projects listed as the following illustration.

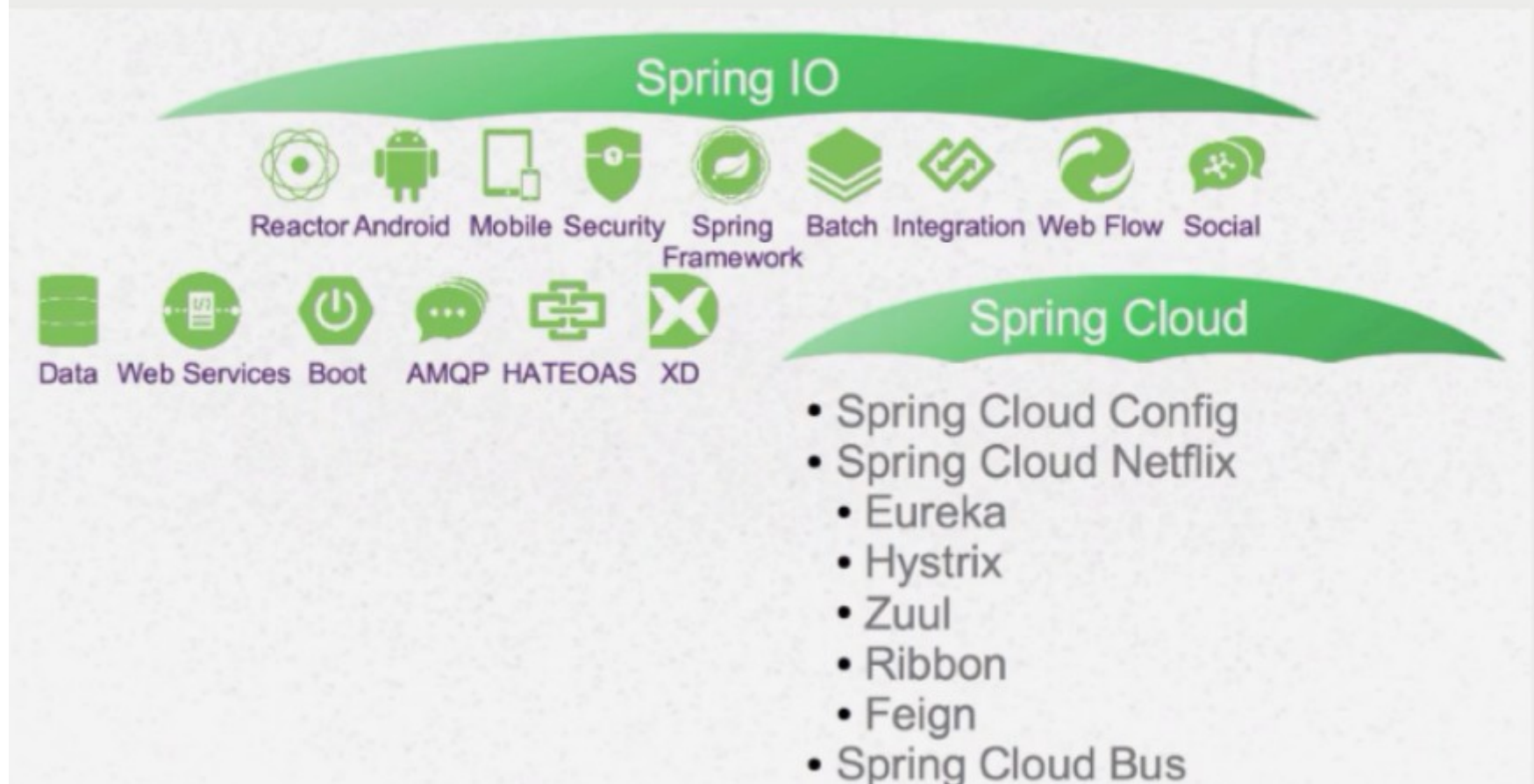


Spring IO (Spring Integration Objects) is the name used for the family of sub-projects of the **Spring**. It is considered as an umbrella and sub-projects are located below such umbrella.

Spring Cloud is a sub-project located in the **Spring IO** Umbrella and it itself is an umbrella, a sub-umbrella.

Introduction to Spring cloud

Spring Cloud is a sub-project located in the **Spring IO** Umbrella and it itself is an umbrella, a sub-umbrella.



Spring Cloud main modules

Below is the list of sub-projects and patterns in the **Spring Cloud**:

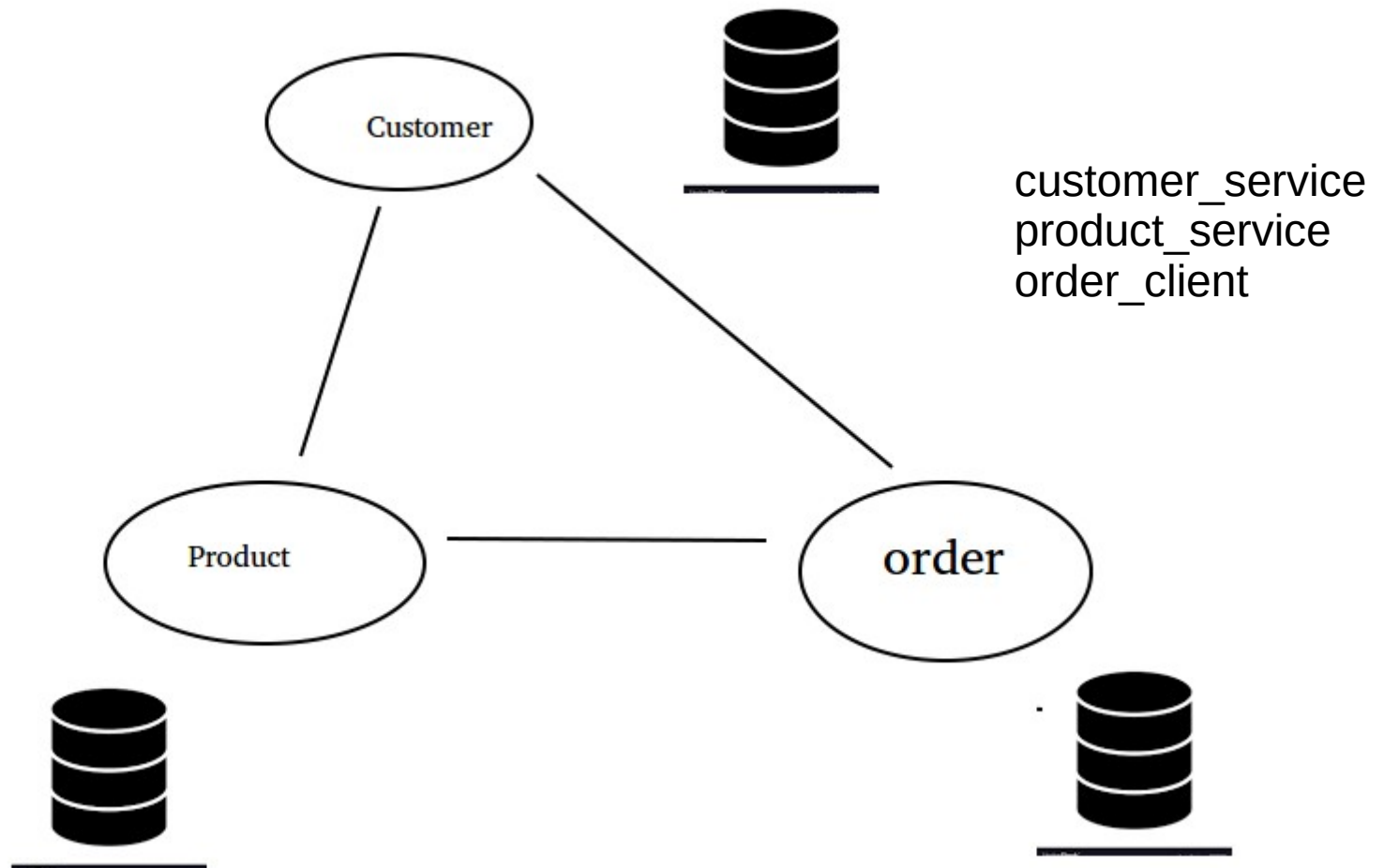
MAIN PROJECTS

- Spring Cloud Config
- Spring Cloud Netflix
- Spring Cloud Bus
- Spring Cloud for Cloud Foundry
- Spring Cloud Cluster
- Spring Cloud Consul
- Spring Cloud Security
- Spring Cloud Sleuth
- Spring Cloud Data Flow
- Spring Cloud Stream
- Spring Cloud Stream Modules
- Spring Cloud Task
- Spring Cloud Zookeeper
- Spring Cloud for Amazon Web Services
- Spring Cloud Connectors
- Spring Cloud Starters
- Spring Cloud CLI

PATTERNS

- Distributed/Versioned Configuration Management
- Service Registration & Discovery
- Routing & Load Balancing
- Fault Tolerance (Circuit Breakers)
- Monitoring
- Concurrent API Aggregation & Transformation
- Distributed messaging

Microservice Example



Creating customer microservices

```
public class Customer {  
    private int id;  
    private String name;  
    private String email;  
}
```

```
public interface CustomerService {  
    public List<Customer> getAllCustomers();  
    public Customer getCustomerById(int id);  
}
```

```
import java.util.ArrayList;  
import java.util.HashMap;  
  
@Service  
public class CustomerServiceImpl implements CustomerService {  
    private static Map<Integer, Customer> customers = new HashMap<Integer, Customer>();  
    static {  
        customers.put(1, new Customer(1, "amit", "amit@gmail.com"));  
        customers.put(2, new Customer(2, "sumit", "sumit@gmail.com"));  
    }  
    @Override  
    public List<Customer> getAllCustomers() {  
        return new ArrayList<Customer>(customers.values());  
    }  
    @Override  
    public Customer getCustomerById(int id) {  
        return customers.get(id);  
    }  
}
```

```
1 server.port=8081  
2 server.servlet.context-path=/customer
```

Creating customer microservices

```
@RestController
public class CustomerController {

    @Autowired
    private CustomerService customerService;

    @GetMapping(value = "/api/customer")
    public List<Customer> getAllBooks() {
        List<Customer> customers = customerService.getAllCustomers();
        return ResponseEntity.ok().body(customers).getBody();
    }

    @GetMapping(value = "/api/customer/{id}")
    public ResponseEntity<Customer> getAnBook(@PathVariable Integer id) {
        Customer book = customerService.getCustomerById(id);
        if (book == null) {
            return new ResponseEntity<Customer>(HttpStatus.NOT_FOUND);
        }

        return new ResponseEntity<Customer>(book, HttpStatus.OK);
    }
}
```

Creating product microservices

```
public class Product {  
    private int id;  
    private String name;  
    private double price;  
}
```

```
public interface ProductService {  
    public List<Product> getAllProducts();  
    public Product getProductById(int id);  
}
```

```
0 @Service  
1 public class ProductServiceImp implements ProductService {  
2     private static Map<Integer, Product> products = new HashMap<Integer, Product>();  
3     static {  
4         products.put(1, new Product(1, "tv", 56));  
5         products.put(2, new Product(1, "laptop", 76));  
6     }  
7 }  
8 @Override  
9 public List<Product> getAllProducts() {  
0  
1     return new ArrayList<Product>(products.values());  
2 }  
3  
4 @Override  
5 public Product getProductById(int id) {  
6     return products.get(id);  
7 }  
8  
9 }
```

```
server.port=8082  
server.servlet.context-path=/product
```

Creating product microservices

```
1 @RestController
2 public class ProductRest {
3
4     @Autowired
5     private ProductService productService;
6
7     @RequestMapping(value = "/api/product", method = RequestMethod.GET, produces =
8         MediaType.APPLICATION_JSON_VALUE)
9     public ResponseEntity<Collection<Product>> getAllBooks() {
10         Collection<Product> products = productService.getAllProducts();
11         return new ResponseEntity<Collection<Product>>(products, HttpStatus.OK);
12     }
13
14     @RequestMapping(value = "/api/product/{id}", method = RequestMethod.GET,
15         produces = MediaType.APPLICATION_JSON_VALUE)
16     public ResponseEntity<Product> getAnBook(@PathVariable Integer id) {
17         Product product = productService.getProductById(id);
18         if (product == null) {
19             return new ResponseEntity<Product>(HttpStatus.NOT_FOUND);
20         }
21         return new ResponseEntity<Product>(product, HttpStatus.OK);
22     }
23 }
```

Creating order microservices

```
public class Order {  
    private long id;  
    private long amount;  
    private Date dateOrder;  
    private Customer customer;  
    private Product product;  
}
```

```
@RestController  
public class OrderRest {  
    @Autowired  
    private RestTemplate restTemplate;  
  
    @RequestMapping(value = "/api/order", method = RequestMethod.GET,  
        produces = MediaType.APPLICATION_JSON_VALUE)  
    public ResponseEntity<Order> submitOrder(@RequestParam("pid") int pid,  
        @RequestParam("cid") int cid) {  
  
        String custUrl="http://localhost:8081/customer/api/customer/"+cid;  
        Customer customer = restTemplate.getForObject(custUrl, Customer.class);  
  
        String productUrl="http://localhost:8082/product/api/product/"+pid;  
        Product product = restTemplate.getForObject(productUrl, Product.class);  
        Order order = new Order();  
        order.setCustomer(customer);  
        order.setProduct(product);  
        order.setId(1);  
        order.setAmount(100);  
        order.setDateOrder(new Date());  
        return new ResponseEntity<Order>(order, HttpStatus.CREATED);  
    }  
}
```

← → ↻ ⓘ localhost:8083/order/api/order?pid=1&cid=1 🔍 ☆ 0

Apps 11 new netwo... Difference bet... Installing and... Unit and Integ... New folder Code Craftsm...

```
{  
  "id": 1, "amount": 100, "dateOrder": "2019-11-17T03:58:03.395+0000",  
  "customer": {  
    "id": 1, "name": "amit", "email": "amit@gmail.com"  
  },  
  "product": {  
    "id": 1, "name": "tv", "price": 56.0  
  }  
}
```


Note: Creating microservices using H2 database

```
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

```
@Entity
@Table(name="item_table")
public class Item {
    @Id
    private int id;
    private String name;
    private int price;
    private int qty;

    @Transient
    private int value;
```

```
1 spring.jpa.show-sql=true
2 spring.h2.console.enabled=true
```

```
insert into item_table(id, name,price,qty) values(1001,'java',500,50);
```

put data in data.sql keep it into resources folder

localhost:8080/h2-console/login.jsp?jsessionId=b23b5254

Apps 11 new netwo... Difference bet... Installing ar

English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:testdb

User Name: sa

Password:

Note: Creating microservices using Java 8

JDK8 approach:

```
@RequestMapping(value = "/user/{id}", method = RequestMethod.GET)
public ResponseEntity<User> getUser(@PathVariable Long id) {
    return Optional
        .ofNullable( userRepository.findOne(id) )
        .map( user -> ResponseEntity.ok().body(user) )           //200 O
        .orElseGet( () -> ResponseEntity.notFound().build() );   //404 N
}
```


Service Discovery

with Eureka and Spring Cloud

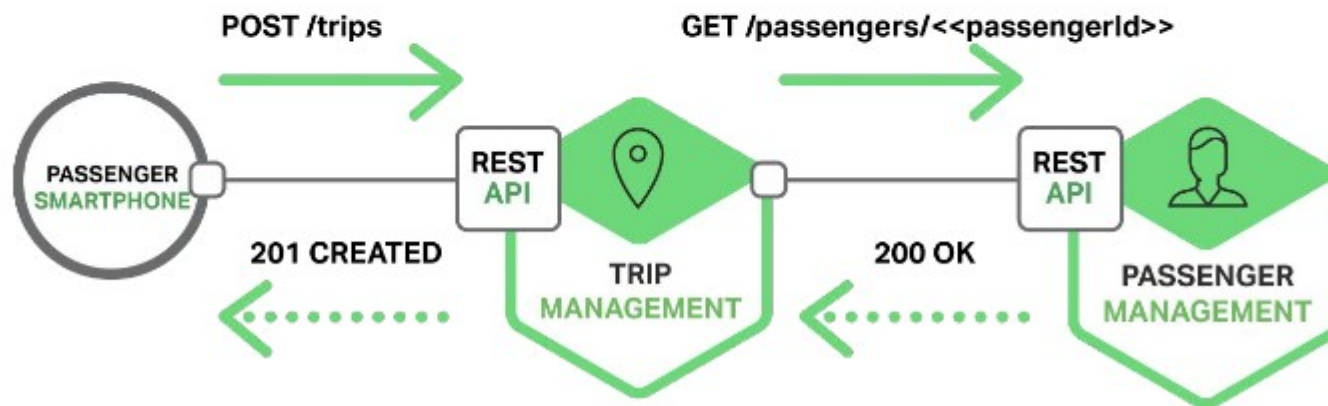
How do services find each other?

What happens if we run multiple instances for a service
AKA yellow pages*

Eureka created by Netflix

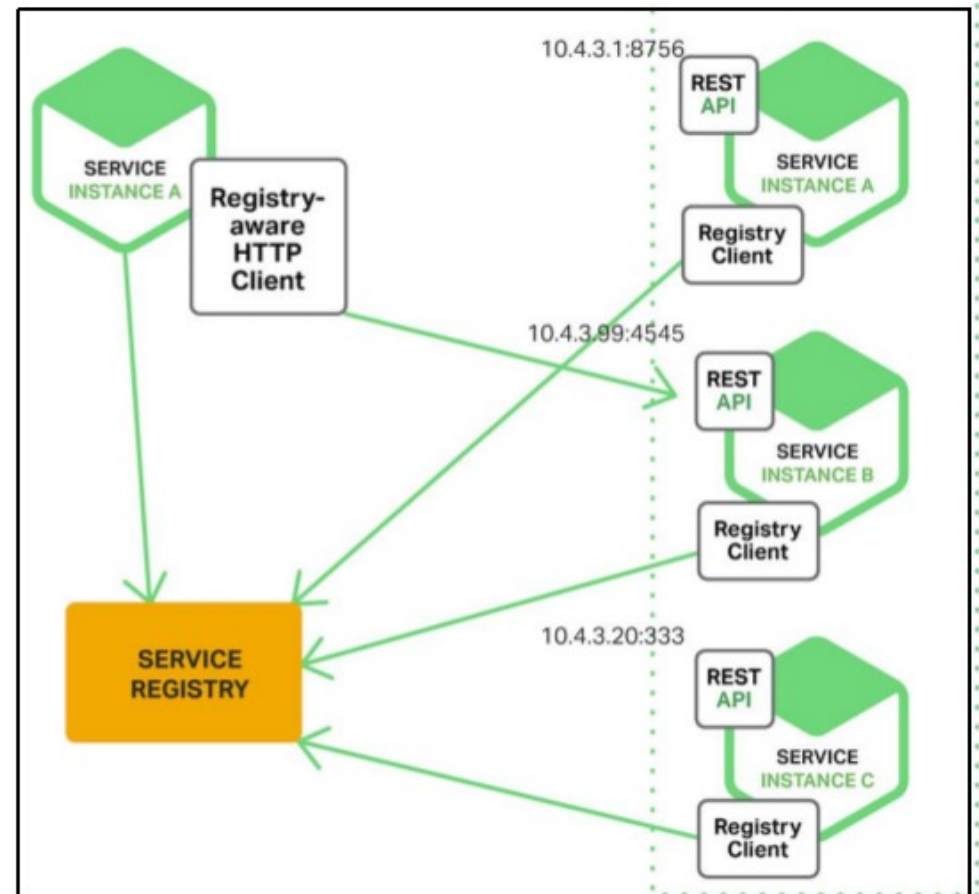
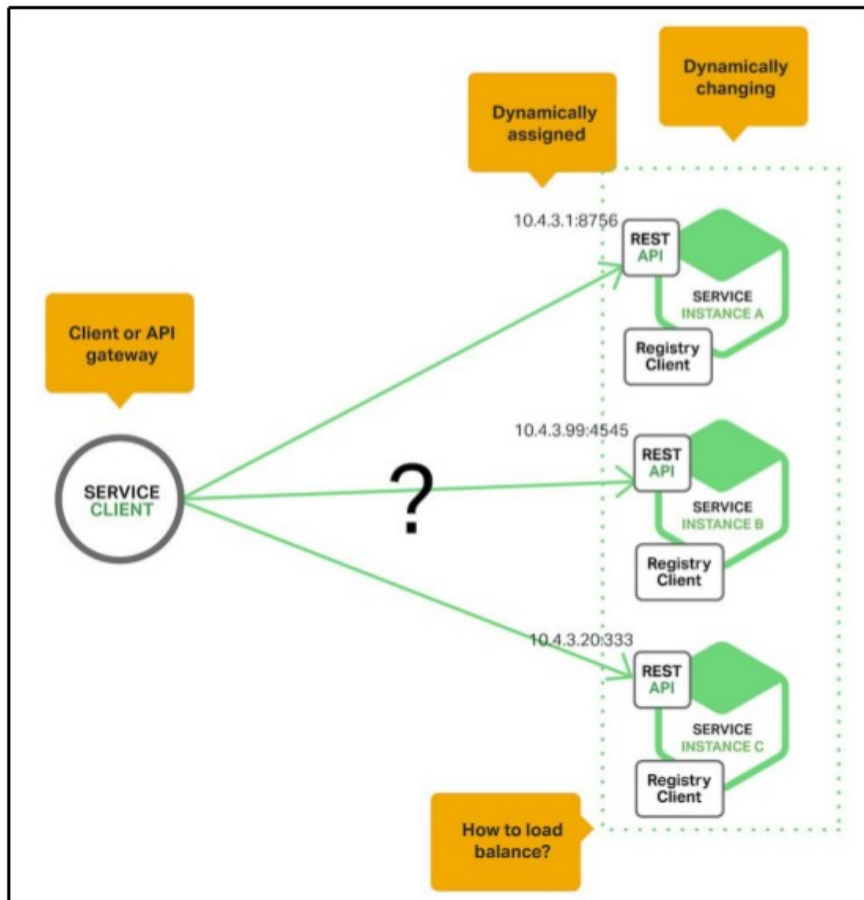
Need of service discovery

To perform communication between services we need know the location of the service(port, host). In traditional applications it's a simple task because services run in a fixed and known location.



In modern applications the services are running in a dynamic environment. A service can have N instances running in N different machines. In this case, to know host and port of each service is very painful.

Need of service discovery





Client-side Load Balancing

- Each service typically deployed as multiple instances for fault tolerance and load sharing.
- But there is problem how to decide which instance to use?
- Netflix Ribbon,
 - it provide several algorithm for Client-Side Load Balancing.
 - Spring provide smart RestTemplate for service discovery and load balancing by using @LoadBalanced annotation with RestTemplate instance.

Configuration service discovery

Boot Version: 2.1.10

Dependencies:

▼ Frequently Used

☒ Spring Boot Actuator

☒ Eureka Server

@EnableEurekaServer


@SpringBootApplication

public class EurekaServerApplication {

```
spring.application.name=discovery-server
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
server.port=8761
```

← → ↻ ⓘ localhost:8761

Apps 11 new netwo... Difference bet... Installing and... Unit and

 **spring Eureka**

System Status

Environment	test
Data center	default

Creating service provider

▼ Spring Cloud Discovery
☒ Eureka Discovery Client
▼ Web
☒ Spring Web

```
@RestController
public class Hello {
    @GetMapping(path="hello")
    public String hello(){
        return "hello from service...";
    }
}
```

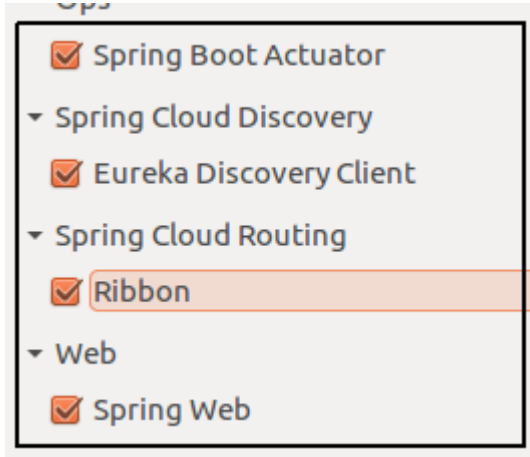
```
7 @EnableEurekaClient
8 @SpringBootApplication
9 public class HelloServiceApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(HelloServiceApplication.class, args);
13     }
14
15 }
```

```
spring.application.name=hello-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
server.port=8085
```

Instances currently registered with Eureka

Application	AMIs	Availability
HELLO_SERVICE	n/a (1)	(1)

Creating client microservice



```
spring.application.name=hello-client
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
server.port=8081
```

```
@EnableEurekaClient
@SpringBootApplication
public class HelloClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args);
    }

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }

}
```

```
@RestController
public class HelloClient {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping(path = "hello_client")
    public String helloClient() {
        return restTemplate.getForObject("http://hello-service/hello", String.class);
    }

}
```




HYSTRIX
DEFEND YOUR APP

Circuit breaker



Its basic function is to interrupt current flow after a fault is detected.

Unlike a fuse, which operates once and then must be replaced, a circuit breaker can be reset (either manually or automatically) to resume normal operation.

Circuit breaker pattern

- Detect something is wrong
- Take temporary steps to avoid the situation getting worse
- Deactivate the “problem” component so that it doesn’t affect downstream components

Microservice fault tolerance

- Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries
- It stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

1) Latency and Fault Tolerance

Stop cascading failures. Fallbacks and graceful degradation. Fail fast and rapid recovery.

Thread and semaphore isolation with circuit breakers.

2) Realtime Operations

Realtime monitoring and configuration changes. Watch service and property changes take effect immediately as they spread across a fleet.

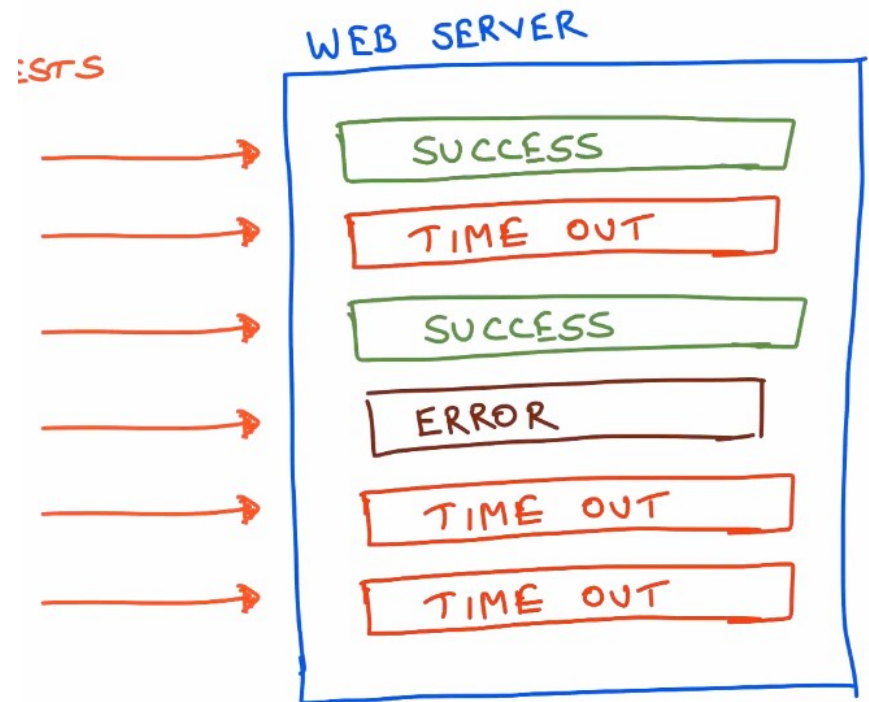
Be alerted, make decisions, affect change and see results in seconds.

3) Concurrency

Parallel execution. Concurrency aware request caching. Automated batching through request collapsing.

Circuit breaker parameter

- Let say one request success, another request timeout, we can not just break the circuit
- What is the point when we should break the circuit, lets say if last 3 request time out i am going to break the circuit, but what if we are getting alternative success and failure we are never going to break the circuit. How to deal?
- Logic to set when circuit should break must be smart to handle all use cases.



Circuit breaker parameter

When does the circuit trip?

- Last n requests to consider for the decision
- How many of those should fail?
- Timeout duration

When does the circuit un-trip?

- How long after a circuit trip to try again?

Last n requests to consider for the decision: 5

How many of those should fail: 3

Timeout duration: 2s

How long to wait (sleep window): 10s

Circuit breaker parameter

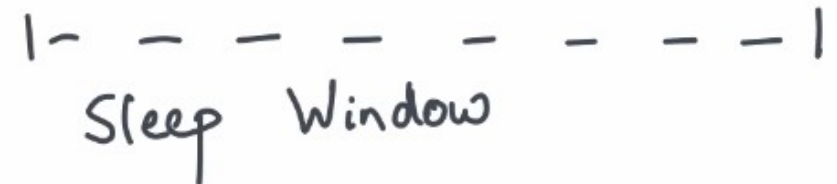
Last n requests to consider for the decision: 5

How many of those should fail: 3

Timeout duration: 2s

How long to wait (sleep window): 10s

Requests to a microservice



Hystrix configuration

☒ Spring Boot Actuator ☒ Eureka Discovery Client ☒ Spring Web
☒ Ribbon

☒ Hystrix ☒ Hystrix Dashboard

```
import org.springframework.boot.SpringApplication;
```

```
1 @EnableHystrix
2 @EnableEurekaClient
3 @SpringBootApplication
4 public class HelloClientApplication {
5
6     public static void main(String[] args) {
7         SpringApplication.run(HelloClientApplication.class, args);
8     }
9 }
```

```
@RestController
public class HelloClient {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping(path = "hello_client")
    @HystrixCommand(fallbackMethod="helloClientHystrix")
    public String helloClient() {
        return restTemplate.getForObject("http://hello-service/hello",String.class);
    }

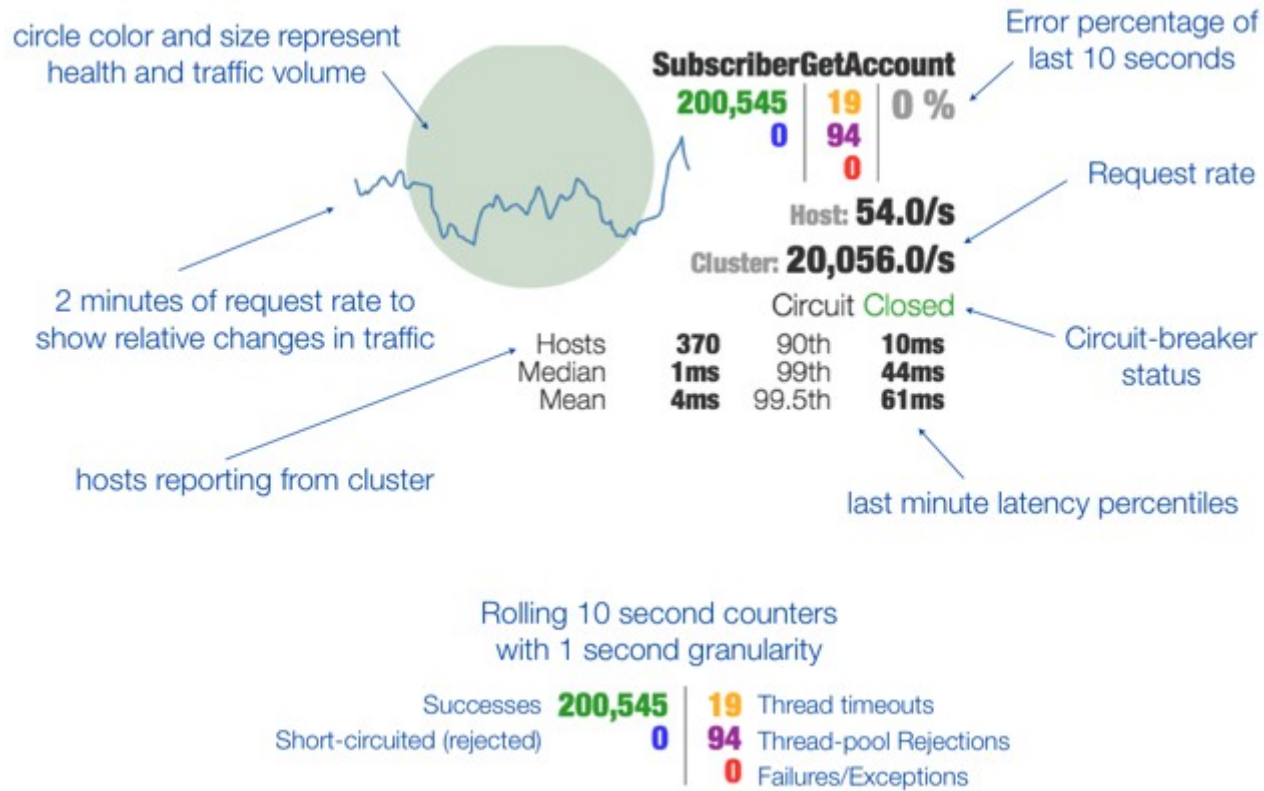
    public String helloClientHystrix() {
        return "hello from fallback service...";
    }
}
```



```
commandProperties = {  
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "2000"),  
    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "5"),  
    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "50"),  
    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "5000")  
}
```

```
@RestController  
public class HelloClient {  
  
    @Autowired  
    private RestTemplate restTemplate;  
  
    // AOP  
    @HystrixCommand(fallbackMethod = "helloDummy", commandProperties =  
        {  
            @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="2000"),  
            @HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="5"),  
            @HystrixProperty(name="circuitBreaker.errorThresholdPercentage", value="50"),  
            @HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds", value="5000")  
        })  
    @GetMapping(path = "client")  
    public String hello() {  
        return restTemplate.getForObject("http://SERVICE-PROVIDER/hello",  
            String.class);  
    }  
  
    public String helloDummy() {  
        return "some dummy response";  
    }  
}
```

Hystrix Dashboard



☒ Spring Boot DevTools☒ Spring Boot Actuator☒ Eureka Discovery Client☒ Hystrix☐ Eureka Server☒ Ribbon☒ Spring Web☒ Hystrix Dashboard

```
@EnableHystrix
@EnableHystrixDashboard
@SpringBootApplication
public class HyConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(HyConsumerApplication.class, args);
    }

}
```

```
1 management.endpoint.hystrix.stream.enabled=true
2 management.endpoints.web.exposure.include=hystrix.stream
```

localhost:8081/hystrix



Hystrix Dashboard

<http://hystrix-app:port/actuator/hystrix.stream>

Cluster via Turbine (default cluster): <http://turbine-hostname:port/turbine.stream>

Cluster via Turbine (custom cluster): [http://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](http://turbine-hostname:port/turbine.stream?cluster=[clusterName])

Single Hystrix App: <http://hystrix-app:port/actuator/hystrix.stream>

Proxy Class

API class

@HystrixCommand

Method

Circuit
breaker
functionality

