Microservices - II

DEVELOPING APPLICATION USING MICROSERVICES

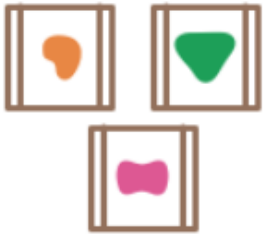
INTRODUCTION TO MICROSERVICES

- Microservices allow large systems to be built up from a number of collaborating components.
- It does at the process level what Spring has always done at the component level
- loosely coupled processes instead of loosely coupled components.

MICROSERVICES VS MONOLITHIC APPLICATION

A monolithic application puts all its functionality into a single process...

A microservices architecture puts each element of functionality into a separate service...



... and scales by replicating the monolith on multiple servers

... and scales by distributing these services across servers, replicating as needed.

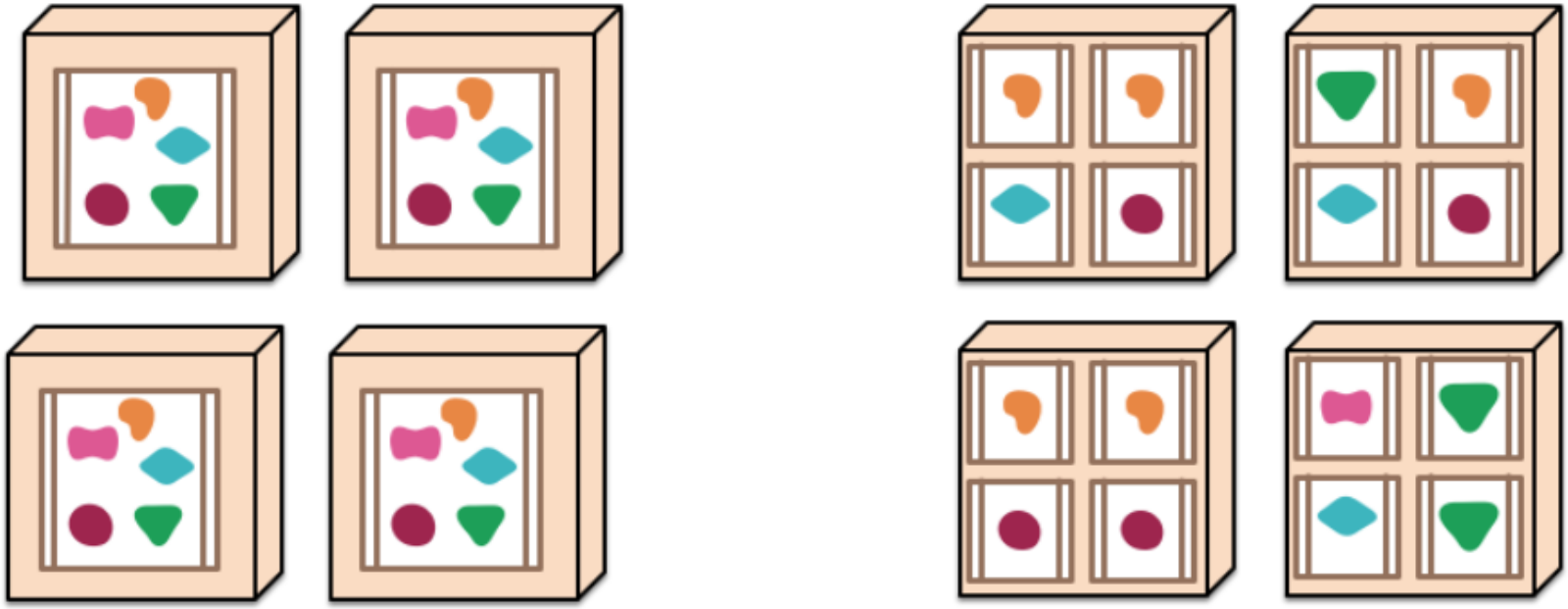


Figure 1: Monoliths and Microservices

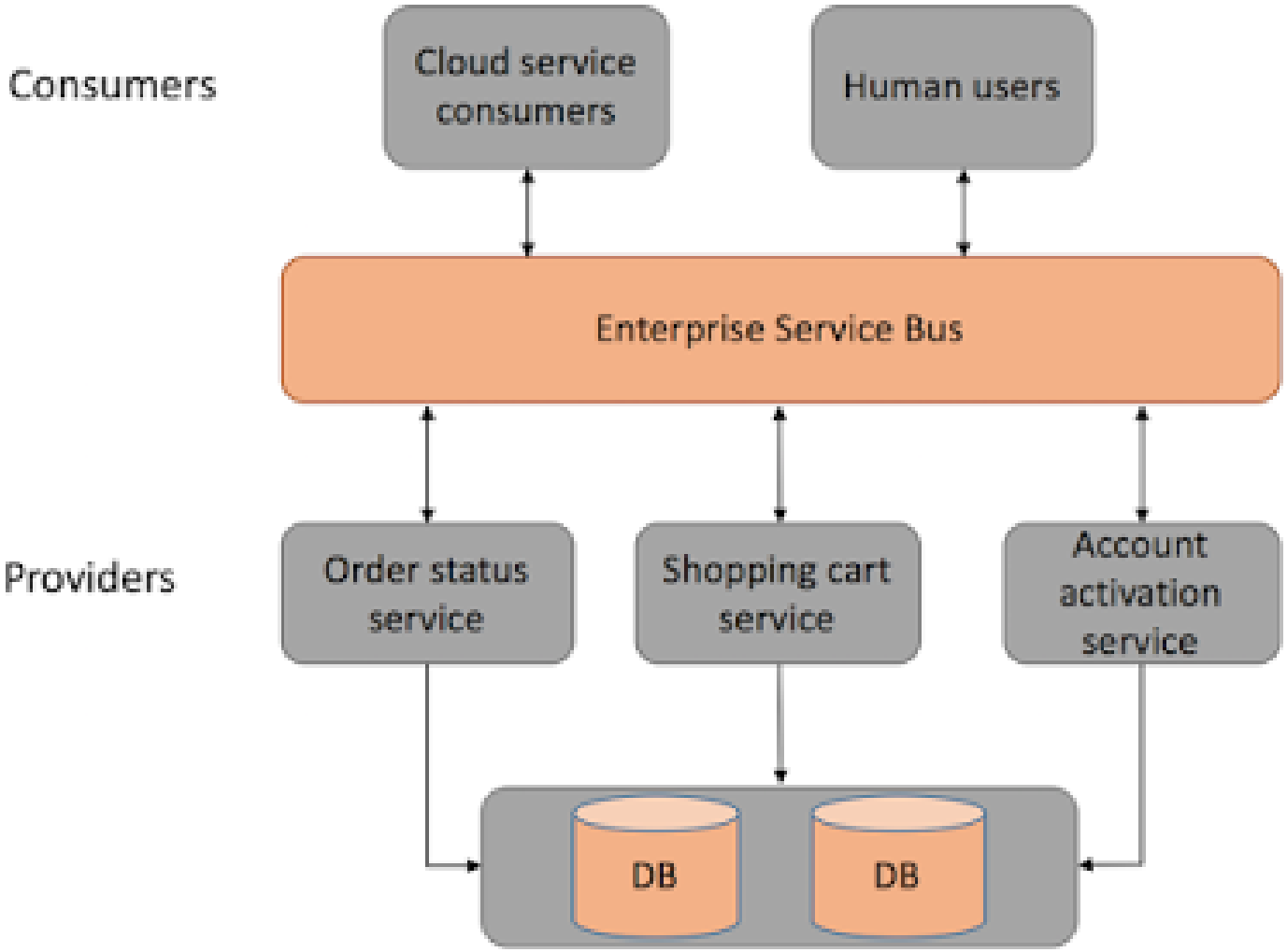
MICROSERVICES VS SOA

One way to avoid hype is to define the buzzwords:

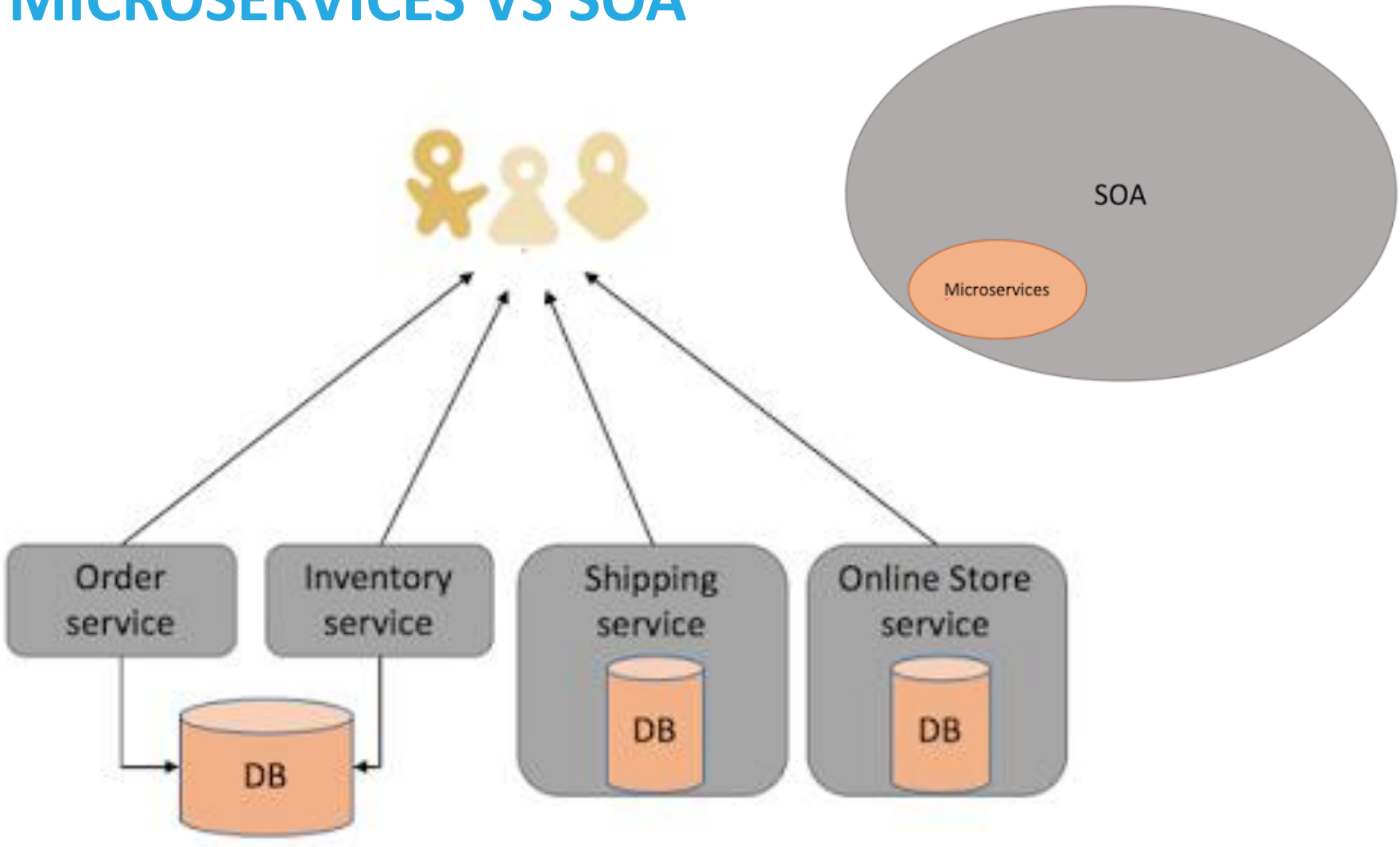
- Service-oriented architecture (SOA): an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network.
- SOA components ***can belong to the same application.***

- Microservices: a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs
- Microservices are independently ***deployable and not belong to the same application.***

MICROSERVICES VS SOA



MICROSERVICES VS SOA



MICROSERVICES VS SOA

Microservices are the kind of SOA we have been talking about for the last decade. Microservices must be independently deployable, whereas SOA services are often implemented in deployment monoliths.

Classic SOA is more platform driven, so microservices offer more choices in all dimensions.

Difference between SOA and microservices lies in the size and scope

MICROSERVICES VS SOA

- If Uber were built with a SOA, their services might be:
  - GetPaymentsAndDriverInformationAndMappingDataAPI
  - AuthenticateUsersAndDriversAPI

- If Uber were built with microservices, their APIs might be more like:
  - SubmitPaymentsService
  - GetDriverInfoService
  - GetMappingDataService
  - AuthenticateUserService
  - AuthenticateDriverService

More APIs, smaller sets of responsibilities.

CHARACTERISTICS OF MICROSERVICES

- Small
- Independent
- Stateless
- Full stack

ARCHITECTURAL PRINCIPLE

- Single Reponsibility
- Domain Driven
- SOA
- Design for Failure
- Automation

12-FACTOR APP PRINCIPLES

1. Codebase
   - *One codebase per service, tracked in revision control; many deploys*
2. Dependencies
   - *Explicitly declare and isolate dependencies*
3. Config
   - Store configuration in the environment
4. Backing Services
   - Treat backing services as attached resources
5. Build, Release, Run
   - Strictly separate build and run stages
6. Processes
   - Execute the app in one or more stateless processes

12-FACTOR APP PRINCIPLES

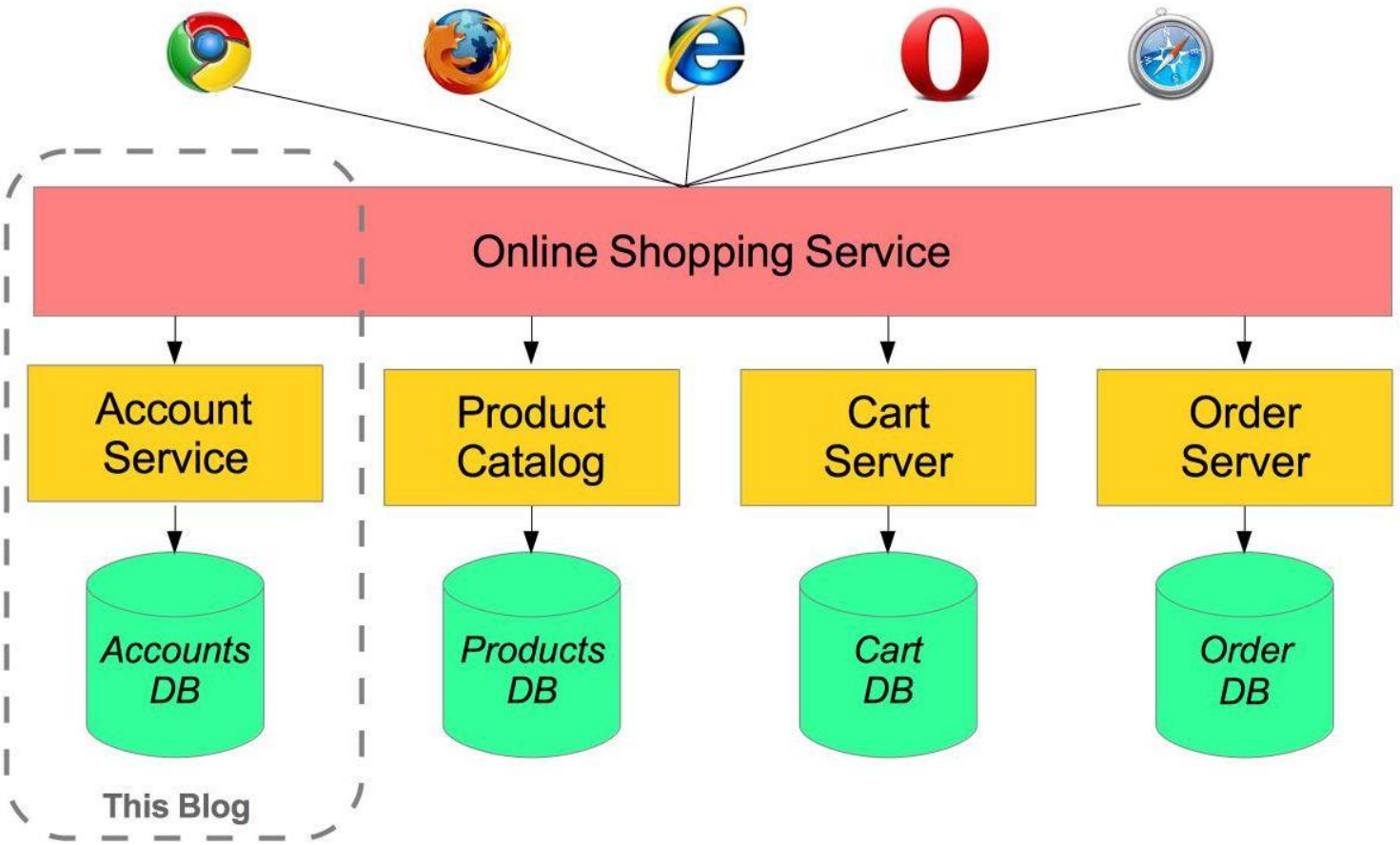
7. Port binding
   - Export services via port binding
8. Concurrency
   - Scale out via the process model
9. Disposability
   - Maximize robustness with fast startup and graceful shutdown
10. Dev/prod parity
    - Keep development, staging, and production as similar as possible
11. Logs
    - Treat logs as event streams
12. Admin processes
    - Run admin/management tasks as one-off processes

HELPER FOR MICROSERVICES

- Spring
- Spring Boot
- Spring Cloud

GET STARTED : MICROSERVICES

- For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts:



GET STARTED : MICROSERVICES

- Here, there are a number of moving parts that you have to setup and configure to build such a system. How to get them working together is not obvious - you need to have good familiarity with Spring Boot since Spring Cloud leverages it heavily, several Netflix or other OSS projects are required and, of course, there is some Spring configuration “magic”!

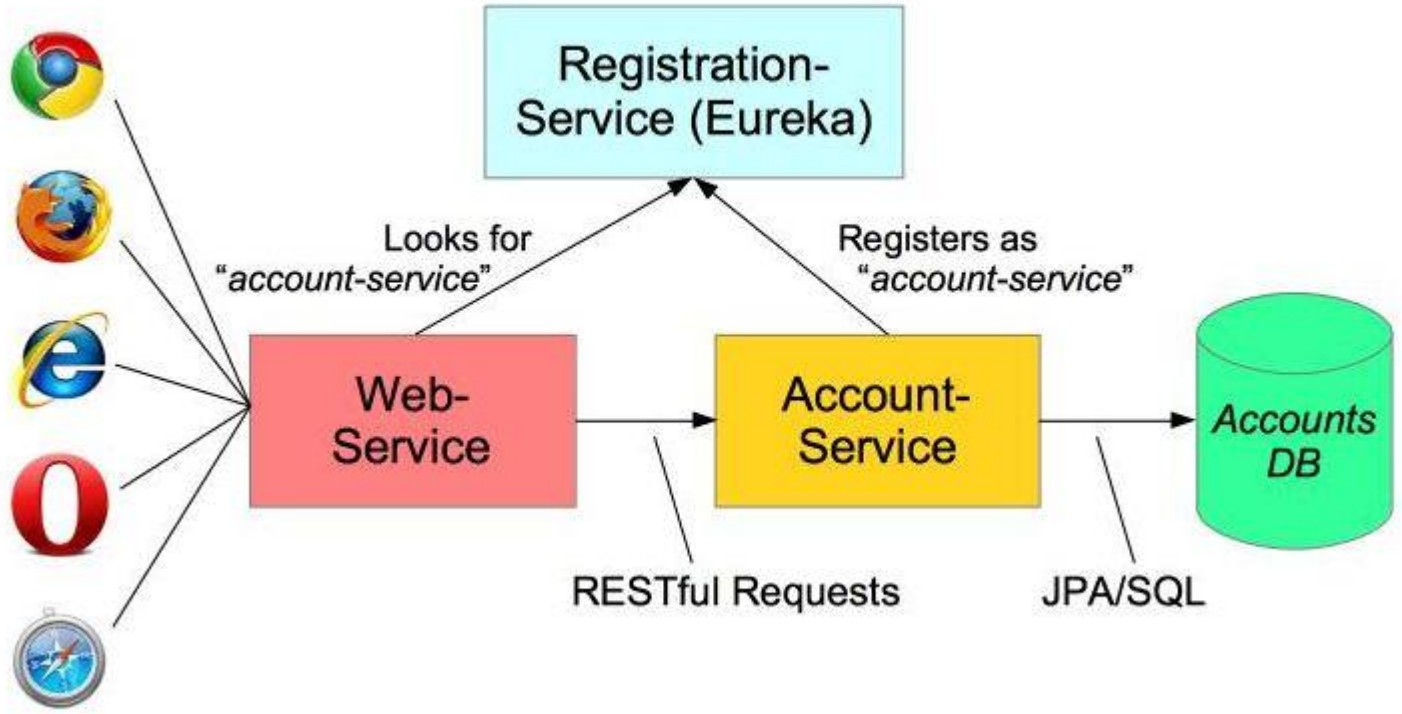
GET STARTED : MICROSERVICES

- Here, there are a number of moving parts that you have to setup and configure to build such a system. How to get them working together is not obvious - you need to have good familiarity with Spring Boot since Spring Cloud leverages it heavily, several Netflix or other OSS projects are required and, of course, there is some Spring configuration “magic”!

- Need to grow from small to big…. ☺

GET STARTED : MICROSERVICES

- Lets implement small part of this system…. the user account service.



GET STARTED : MICROSERVICES

- **Service Registration**
- When you have multiple processes working together they need to find each other like Java's RMI mechanism. Microservices has the same requirement.
- Netflix created a open source registration server called Eureka.
- Spring has incorporated this discovery server into Spring Cloud.
- Here is the *complete* discovery-server application:

GET STARTED : MICROSERVICES

- **Service Registration**
- Here is the *complete* discovery-server application:

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistrationServer {

  public static void main(String[] args) {
    // Tell Boot to look for registration-server.yml
    System.setProperty("spring.config.name", "registration-server");
    SpringApplication.run(ServiceRegistrationServer.class, args);
  }
}
```

GET STARTED : MICROSERVICES

- **Getting Spring Cloud and Eureka Server as Spring Boot Starter**

```
<dependency>
    <!-- Spring Cloud starter -->
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter</artifactId>
</dependency>

<dependency>
    <!-- Eureka for service registration -->
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

GET STARTED : MICROSERVICES

This application looks for `registration-server.properties` or `registration-server.yml`.
Here is the relevant configuration from `registration-server.yml`:

```
# Configure this Discovery Server
eureka:
  instance:
    hostname: localhost
  client:  # Not a client, don't register with yourself
    registerWithEureka: false
    fetchRegistry: false

server:
  port: 1111   # HTTP (Tomcat) port
```

By default Eureka runs on port 8761, but here we will use port 1111 instead

GET STARTED : MICROSERVICES

- **Running Eureka Registry Server using following command**

Java -jar target/microservice-demo-1.1.0.RELEASE.jar registration

- **Open Server using** http://localhost:1111 link in browser.
- **Access Eureka DASHBOARD.**

- **Alternative to Eureka :- Consul**

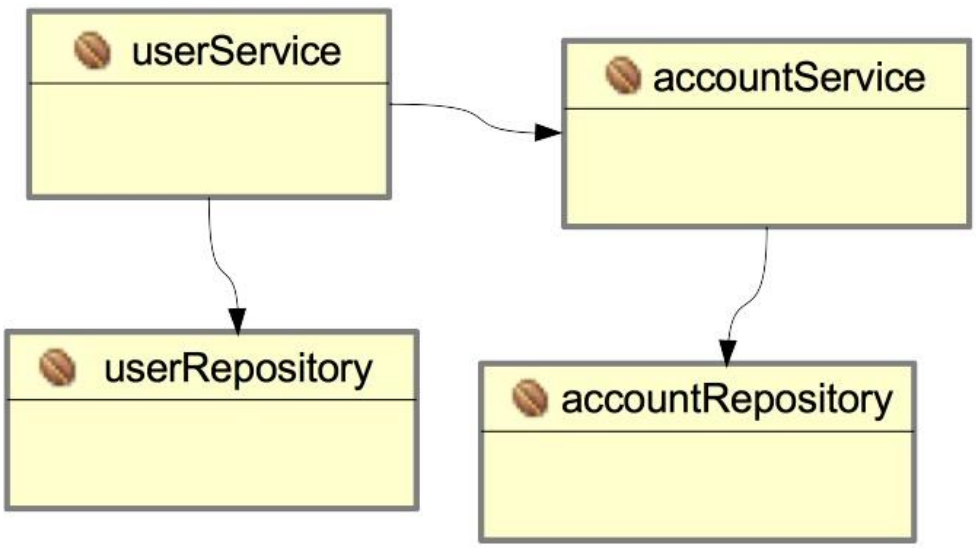
CREATING MICROSERVICES

Creating a Microservice: *Account-Service*

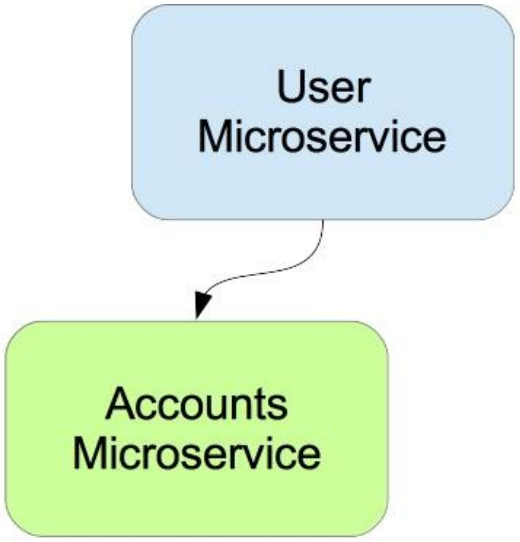
- A microservice is a stand-alone process that handles a well-defined requirement.
- When configuring applications with Spring we emphasize Loose Coupling and Tight Cohesion.
- But now we are applying them, not to interacting components (Spring Beans), but to interacting processes.
- **Next, we have to** implement a simple Account management microservice that uses Spring Data to implement a JPA AccountRepository and Spring REST to provide a RESTful interface to account information.

CREATING MICROSERVICES

Creating a Microservice: *Account-Service*



Spring Bean Dependencies



Interacting Processes

CREATING MICROSERVICES

Implementation of Microservice: *Account-Service*

```
@EnableAutoConfiguration
@EnableDiscoveryClient
@Import(AccountsWebApplication.class)
public class AccountsServer {

    @Autowired
    AccountRepository accountRepository;

    public static void main(String[] args) {
        // Will configure using accounts-server.yml
        System.setProperty("spring.config.name", "accounts-server");

        SpringApplication.run(AccountsServer.class, args);
    }
}
```

CREATING MICROSERVICES

- What makes it special is that it registers itself with the *discovery-server* at start-up.
- **@EnableAutoConfiguration** - defines this as a Spring Boot application.
- **@EnableDiscoveryClient** - this enables service registration and discovery. In this case, this process registers itself with the *discovery-server* service using its application name.
- **@Import(AccountsWebApplication.class)** - this Java Configuration class sets up everything else.

CREATING MICROSERVICES

- **Here is master mind →**

```
# Spring properties
spring:
  application:
     name: accounts-service

# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/

# HTTP Server
server:
  port: 2222   # HTTP (Tomcat) port
```

CREATING MICROSERVICES

- **Running Account Service**

  java -jar target/microservice-demo-1.1.0.RELEASE.jar accounts

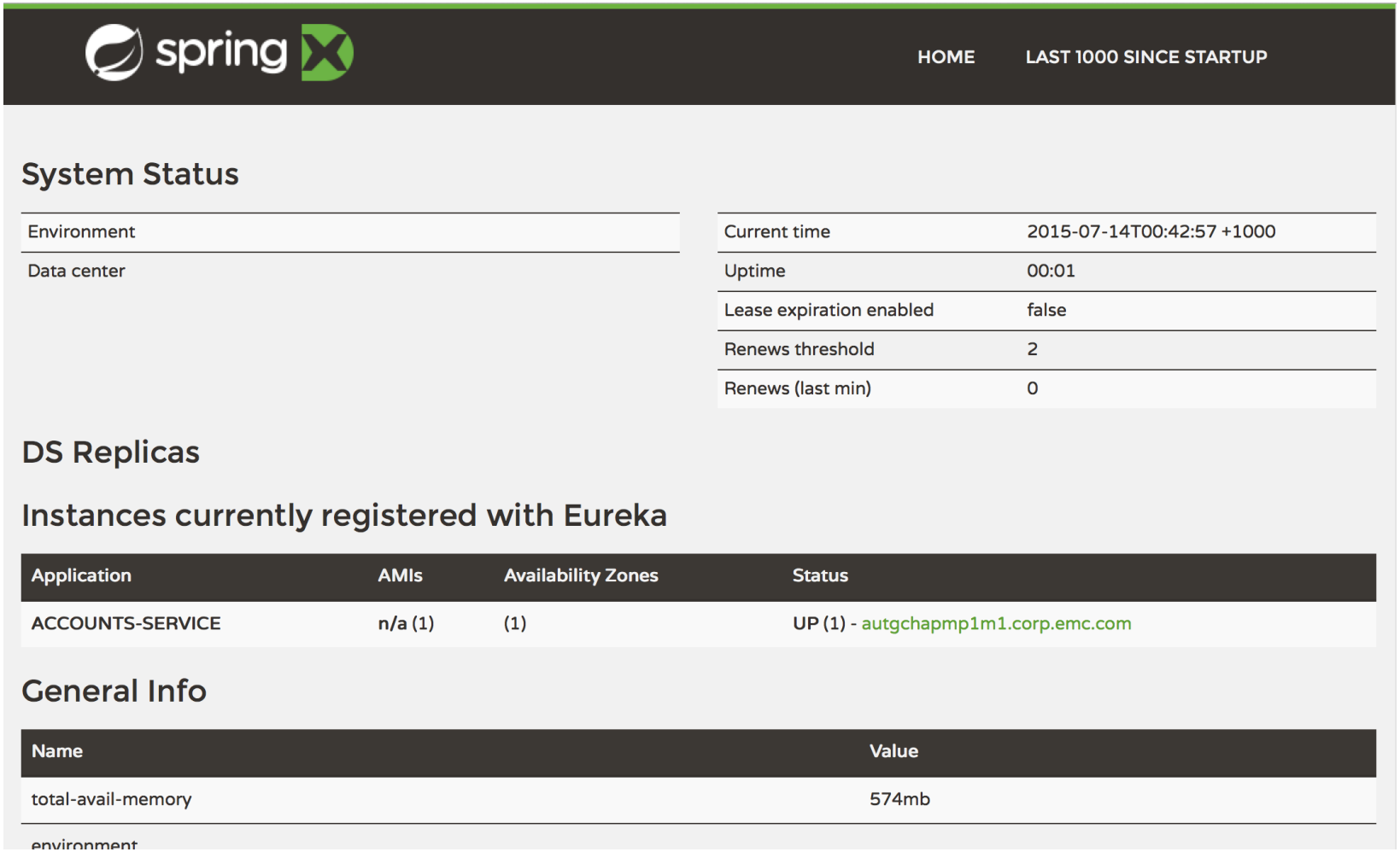
- Refresh the dashboard http://localhost:1111 and you should see the ACCOUNTS-SERVICE listed under Applications.
- For more detail, go here: http://localhost:1111/eureka/apps/

- Alternatively go to

http://localhost:1111/eureka/apps/ACCOUNTS-SERVICE and see just the details for *AccountsService* - if it's not registered you will get a 404.

CREATING MICROSERVICES

- **Running Account Service**



CREATING MICROSERVICES

- **Registration Id:** A process (microservice) registers with the discovery-service using a unique id.

- If another process registers with the *same* id, it is treated as a restart (for example some sort of failover or recovery) and the first process registration is discarded. This gives us the fault-tolerant system we desire.

- To run multiple instances of the *same* process (for load-balancing and resilience) they need to register with a unique id

CREATING MICROSERVICES

Accessing the Microservice: *Web-Service*

- To consume a RESTful service, Spring provides the RestTemplate class. This allows you to send HTTP requests to a RESTful server and fetch data in a number of formats - such as JSON and XML.

- A microservice (discovery) client can use a RestTemplate and Spring will automatically configure it to be microservice aware.

- Note: The Accounts microservice provides a RESTful interface over HTTP, but any suitable protocol could be used. Messaging using AMQP or JMS is an obvious alternative.

ENCAPSULATING MICROSERVICE ACCESS

```
@Service
public class WebAccountsService {

    @Autowired        // NO LONGER auto-created by Spring Cloud (see below)
    @LoadBalanced
    protected RestTemplate restTemplate;

    protected String serviceUrl;

    public WebAccountsService(String serviceUrl) {
        this.serviceUrl = serviceUrl.startsWith("http") ?
                serviceUrl : "http://" + serviceUrl;
    }

    public Account getByNumber(String accountNumber) {
        Account account = restTemplate.getForObject(serviceUrl
                + "/accounts/{number}", Account.class, accountNumber);

        if (account == null)
            throw new AccountNotFoundException(accountNumber);
        else
            return account;
    }
    ...
```

ACCESSING THE MICROSERVICE

- The serviceUrl is provided by the main program to the WebAccountController which in turn passes it to the WebAccountService

- A RestTemplate instance is thread-safe and can be used to access any number of services in different parts of your application (for example, I might have a CustomerService wrapping the same RestTemplate instance accessing a customer data microservice).

ACCESSING THE MICROSERVICE

```
@SpringBootApplication
@EnableDiscoveryClient
@ComponentScan(useDefaultFilters=false)  // Disable component scanner
public class WebServer {

    public static void main(String[] args) {
        // Will configure using web-server.yml
        System.setProperty("spring.config.name", "web-server");
        SpringApplication.run(WebServer.class, args);
    }

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

    @Bean
    public WebAccountsController accountsController() {
         // 1. Value should not be hard-coded, just to keep things simple
         //        in this example.
         // 2. Case insensitive: could also use: http://accounts-service
         return new WebAccountsController
                        ("http://ACCOUNTS-SERVICE");  // serviceUrl
```

ACCESSING THE MICROSERVICE

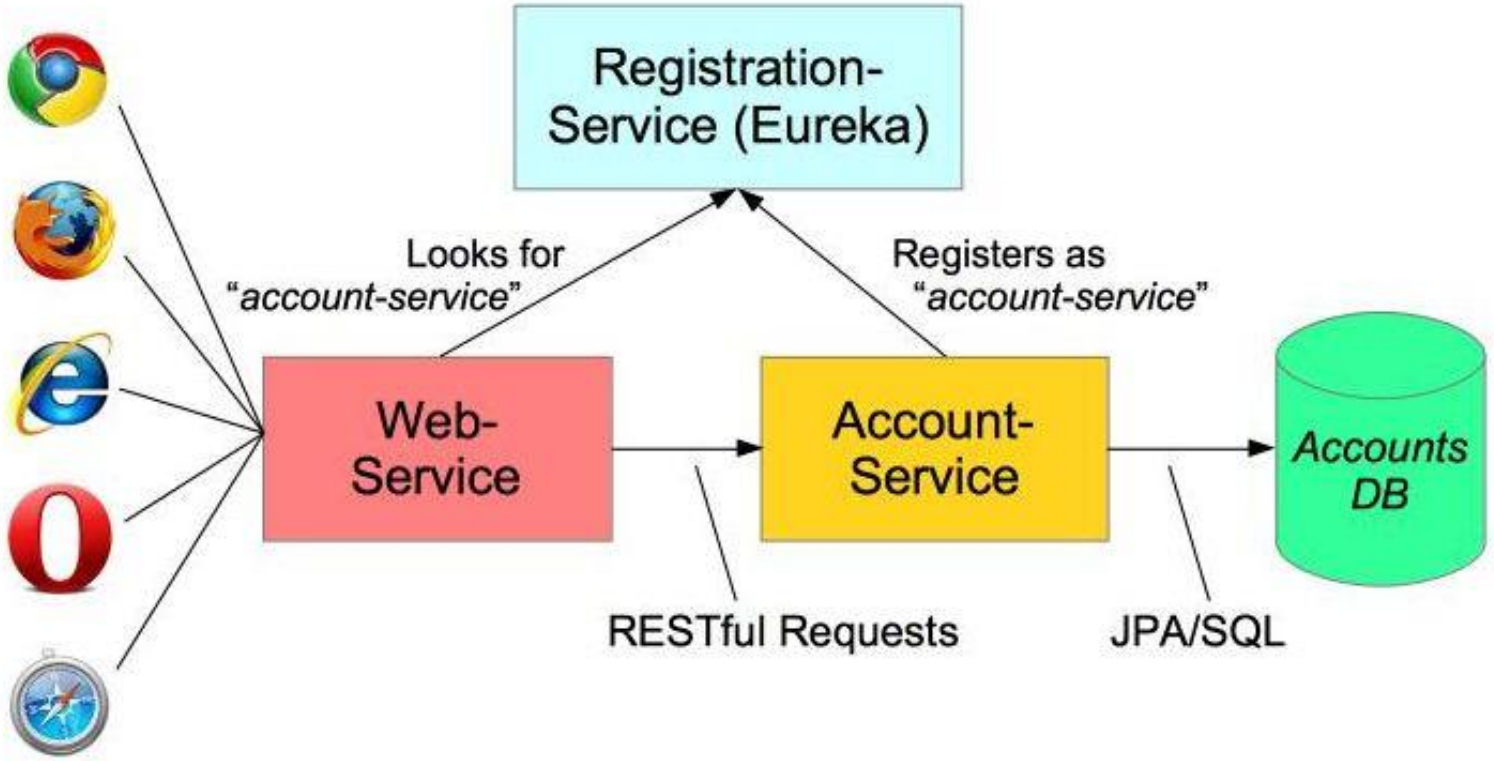
- web-server.yml

```yaml
# Spring Properties
spring:
  application:
     name: web-service

# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/

# HTTP Server
server:
  port: 3333   # HTTP (Tomcat) port
```

RUNNING THE APPLICATION



RUNNING THE APPLICATION

- Run the three servers in order:
  - *RegistrationService*,
  - *AccountsService*
  - and *WebService*

RUNNING THE APPLICATION

- **From CMD**
  - In each window, change to the directory where you cloned the demo
  - In the first window, build the application using mvn clean package
  - In the same window run: java -jar target/microservice-demo-1.1.0.RELEASE.jar registration and wait for it to start up
  - Switch to the second window and run: java -jar target/microservice-demo-1.1.0.RELEASE.jar accounts and again wait for it to start up
  - In the third window run: java -jar target/microservice-demo-1.1.0.RELEASE.jar web
  - In your favorite browser open the same two links: http://localhost:1111 and http://localhost:3333

LOAD BALANCING - RIBBON

- @LoadBalanced
- The annotated RestTemplate should use a RibbonLoadBalancerClient for interacting with your service(s).

- In turn, this allows you to use "logical identifiers" for the URLs you pass to the RestTemplate. These logical identifiers are typically the name of a service. For example:

  - *restTemplate.getForObject("http://some-service-name/user/{id}", String.class, 1);*
- where some-service-name is the logical identifier.
  - *@RibbonClient*
- Used for configuring your Ribbon client(s).

LOAD BALANCING - RIBBON

- Is @RibbonClient required?

- No! If you're using Service Discovery and you're ok with all of the default Ribbon settings, you don't even need to use the @RibbonClient annotation.

- When should I use @RibbonClient?
  - There are at least two cases where you need to use @RibbonClient
  - You need to customize your Ribbon settings for a particular Ribbon client
  - You're not using any service discovery

LOAD BALANCING - RIBBON

- Customizing your Ribbon settings:
- Define a @RibbonClient
  - *@RibbonClient(name = "some-service", configuration = SomeServiceConfig.class)*
- name - set it to the same name of the service you're calling with Ribbon but need additional customizations for how Ribbon interacts with that service.
- configuration - set it to an @Configuration class with all of your customizations defined as @Beans. Make sure this class is not picked up by @ComponentScan otherwise it will override the defaults for ALL Ribbon clients.

LOAD BALANCING - RIBBON

- Using Ribbon without Service Discovery
- If you're not using Service Discovery, the name field of the @RibbonClient annotation will be used to prefix your configuration in the application.properties as well as "logical identifier" in the URL you pass to RestTemplate.

- Define a @RibbonClient
  - *@RibbonClient(name = "myservice")*
- then in your application.properties

  - myservice.ribbon.eureka.enabled=false
  - myservice.ribbon.listOfServers=http://localhost:5000, http://localhost:5001

RUNNING THE APPLICATION

- **From CMD**
  - You should see servers being registered in the log output of the first (registration) window.
  - As you interact with the web-application (http://localhost:3333) you should logging appear in the second and third windows.
  - **Load Balancing**
  - In a new window, run up a second account-server using HTTP port 2223:
  - java -jar target/microservice-demo-0.0.1-SNAPSHOT.jar accounts 2223
  - Allow it to register itself
  - Kill the first account-server and see the web-server switch to using the new account-server - no loss of service.

SPRING CLOUD

- LoadBalancer
- CircuitBroker
- API Gateway

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.
- Spring Cloud supports both Eureka and Consul
- Netflix has a Eureka-aware client-side load-balancing client called Ribbon that Spring Cloud integrates extensively.
- Ribbon is a client library with built-in software load balancers

SPRING CLOUD

- API Gateway
- It is a single entry point into the system, used to handle requests by routing them to the appropriate backend service or by invoking multiple backend services and aggregating the results.
- Also, it can be used for authentication, insights, stress and canary testing, service migration, static response handling, active traffic management.
- @EnableZuulProxy

SPRING CLOUD

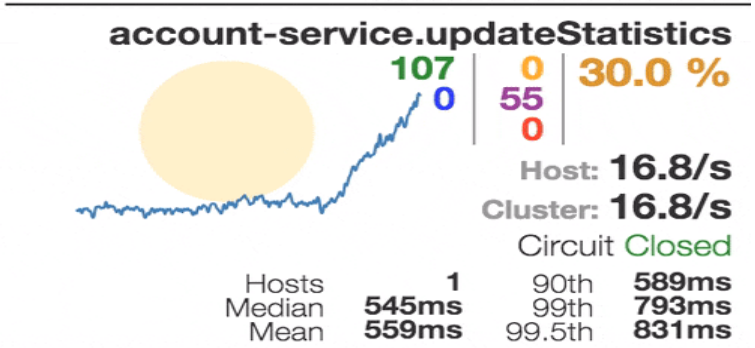
- **Load balancer,**
- **Ribbon**
- Ribbon is a client side load balancer which gives you a lot of control over the behaviour of HTTP and TCP clients. Compared to a traditional load balancer, there is no need in additional hop for every over-the-wire invocation - you can contact desired service directly.

SPRING CLOUD

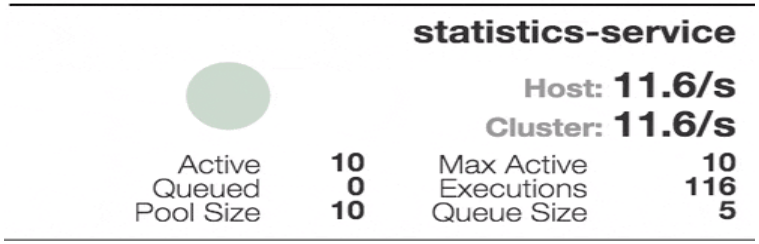
- **Hystrix = Circuit breaker**
- Hystrix is the implementation of Circuit_Breaker_pattern, which gives a control over latency and failure from dependencies accessed over the network.
- The main idea is to stop cascading failures in a distributed environment with a large number of microservices. That helps to fail fast and recover as soon as possible - important aspects of fault-tolerant systems that self-heal.
- Besides circuit breaker control, with Hystrix you can add a fallback method that will be called to obtain a default value in case the main command fails.

SPRING CLOUD

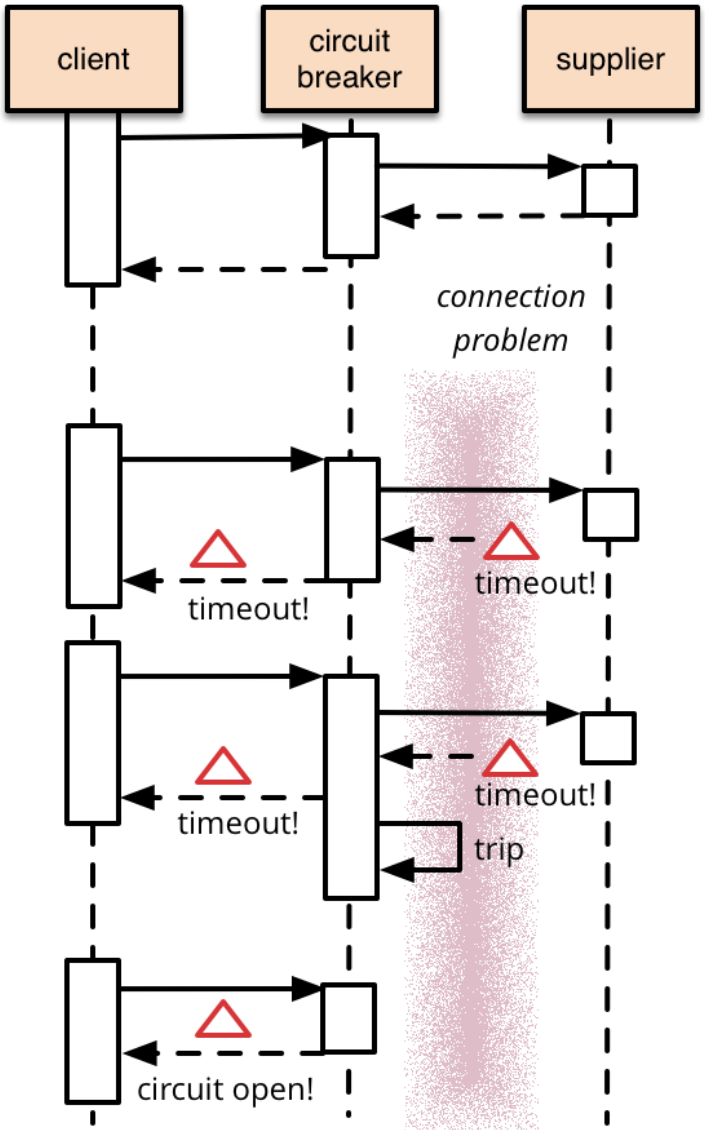
Circuit



Thread Pools



SPRING CLOUD



WHAT IS FEIGN?

- Feign is a Java to HTTP client binder. Feign Simplifies the HTTP API Clients using declarative way.

- Feign is a library for creating REST API clients in a declarative way. it makes writing web service clients easier. Developers can use declarative annotations to call rest services instead of writing repetitive boilerplate code.

- Spring Cloud OpenFeign provides OpenFeign integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment.

USING FEIGN OVER RESTTEMPLATE

- 1.URLs are not hardcoded.
- 2.you don't have to write unit test cases for feign as there is no code to test however you have to write integration tests.
- 3.we can use Eureka Client ID instead of the URL.
- 4.Feign handled the actual code.
- 5.Feign integrates with Ribbon and Eureka Automatically.
- 6.Feign provides a very easy way to call RESTful services.

HTTP CLIENT USING RESTTEMPLATE

```
@GetMapping
  public List<User> getAllUsers() {
    System.out.println("Calling User Service using Feign Client!!");
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<User>> response = restTemplate.exchange(
        "http://localhost:8080/user/",
        HttpMethod.GET,          null,
        new ParameterizedTypeReference<List<User>>() {
        });
    List<User> users = response.getBody();
    return users;
  }
  @GetMapping("{id}")
  public User getUserById(@PathVariable("id") int id) {
    Map<String, String> uriParams = new HashMap<String, String>();
    uriParams.put("id", String.valueOf(id));
    URI uri = UriComponentsBuilder
        .fromUriString("http://localhost:8080/user/{id}")
        .buildAndExpand(uriParams)
        .toUri();
    System.out.println(uri);
    RestTemplate restTemplate = new RestTemplate();
    User forEntity = restTemplate.getForObject(uri, User.class);
    return forEntity;
  }
```

HTTP CLIENT USING RESTTEMPLATE

```
@PostMapping
  public ResponseEntity addUser(@RequestBody User user) {
    System.out.println("Add user");
    System.out.println(user.toString());
    RestTemplate restTemplate = new RestTemplate();
    HttpEntity<User> request = new HttpEntity<>(user);
    ResponseEntity exchange = restTemplate
        .exchange("http://localhost:8080/user/", HttpMethod.POST, request, String.class);
    return exchange;
  }
  @DeleteMapping("{id}")
  public ResponseEntity deleteUser(@PathVariable int id) {
    System.out.println("delete user");
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", String.valueOf(id));
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.delete("http://localhost:8080/user/{id}", params);
    return new ResponseEntity("User Deleted successfully", HttpStatus.OK);
  }
```

HTTP CLIENT USING FEIGN CLIENT

```
@FeignClient(name = "User", url = "http://localhost:8080")
public interface UserClient {

    @RequestMapping(method = RequestMethod.GET, value = "/user")
    List<User> getAllUsers();

    @RequestMapping(method = RequestMethod.GET, value = "/user/{id}")
    User getUser(@PathVariable("id") int id);

    @RequestMapping(method = RequestMethod.DELETE, value = "/user/{id}")
    ResponseEntity deleteUser(@PathVariable("id") int id);

    @RequestMapping(method = RequestMethod.POST, value = "/user/")
    ResponseEntity addUser(@RequestBody User user);

    @RequestMapping(method = RequestMethod.PUT, value = "/user/")
    ResponseEntity updateUser(User user);}
```

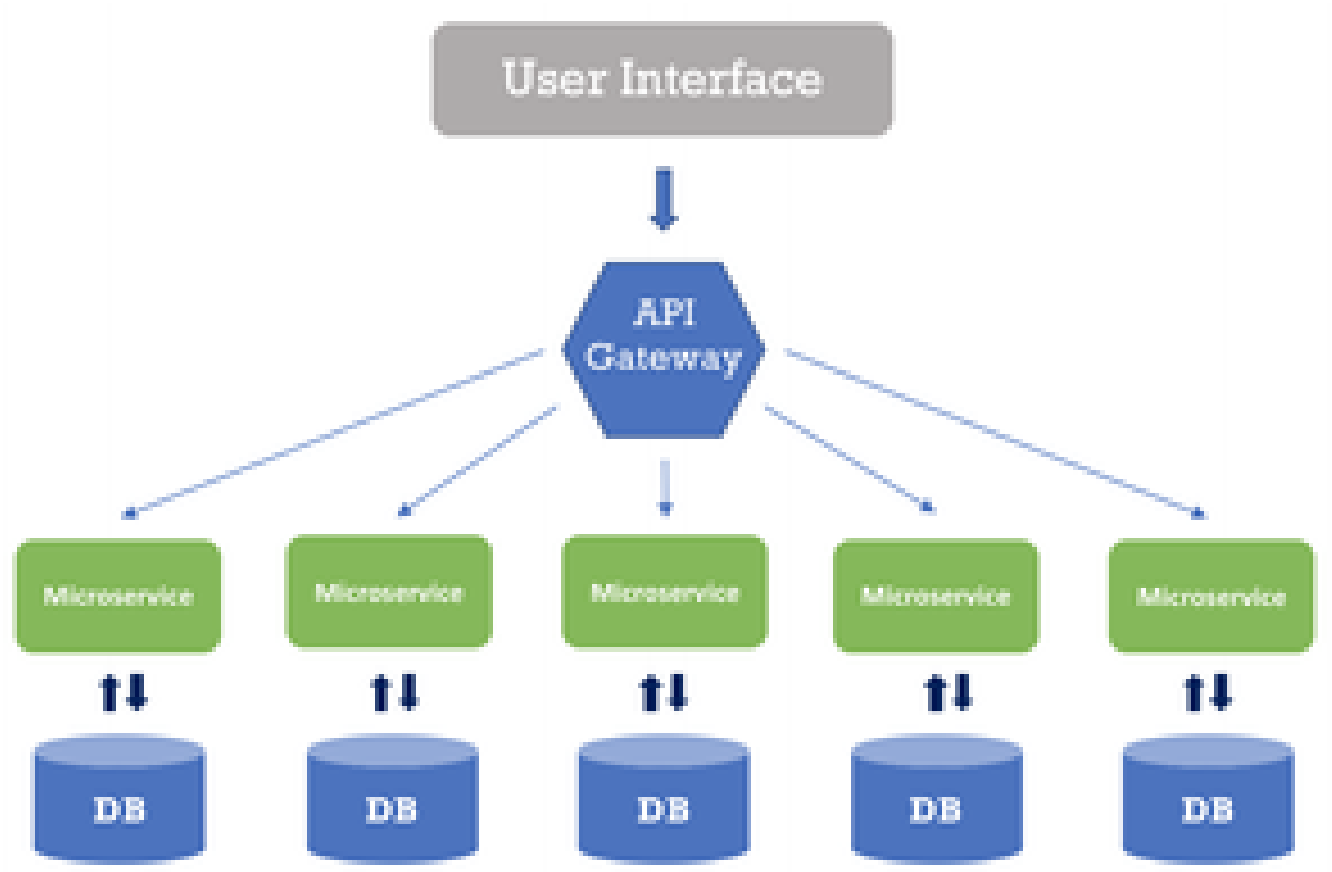
API GATEWAY

- API gateway is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service.

- It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

- The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice.

API GATEWAY


User Interface
API Gateway
Microservice
Microservice
Microservice
Microservice
Microservice
DB
DB
DB
DB
DB

API GATEWAY - ZUUL

- API gateway is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service.

- It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

- The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice.

ZIPKIN – DISTRIBUTED TRACING

- Zipkin, based on Google Dapper and initially developed by Twitter, is a Java-based application that is used for distributed tracing and identifying latency issues. Unique identifiers are attached to each request which are then passed downstream through the different waypoints, or services.

- Zipkin is very efficient tool for distributed tracing in microservices ecosystem. Distributed tracing, in general, is latency measurement of each component in a distributed transaction where multiple microservices are invoked to serve a single business usecase.

ZIPKIN – DISTRIBUTED TRACING

- Zipkin, based on Google Dapper and initially developed by Twitter, is a Java-based application that is used for distributed tracing and identifying latency issues.

- Unique identifiers are attached to each request which are then passed downstream through the different waypoints, or services.

- Zipkin is very efficient tool for distributed tracing in microservices ecosystem.

- Distributed tracing, in general, is latency measurement of each component in a distributed transaction where multiple microservices are invoked to serve a single business usecase.

ZIPKIN – DISTRIBUTED TRACING

- If you are troubleshooting latency problems or errors in ecosystem, you can filter or sort all traces based on the application, length of trace, annotation, or timestamp.

- By analyzing these traces, you can decide which components are not performing as per expectations, and you can fix them.

ZIPKIN – DISTRIBUTED TRACING

- It has 4 modules
- **Collector** –
  - Once any component sends the trace data arrives to Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.
- **Storage** –
  - This module store and index the lookup data in backend. Cassandra, ElasticSearch and MySQL are supported.
- **Search** –
  - This module provides a simple JSON API for finding and retrieving traces stored in backend. The primary consumer of this API is the Web UI.
- **Web UI** –
  - A very nice UI interface for viewing traces.

ZIPKIN – INSTALLATION

- download the latest Zipkin server from maven repository and run the executable jar file using below command.
  - ***java -jar zipkin-server-1.30.3-exec.jar***

- Start Web UI at **http://localhost:9411/zipkin/**

- To install Zipkin in spring boot application, we need to add Zipkin starter dependency in spring boot project.

- <dependency>
- <groupId>org.springframework.cloud</groupId>
- <artifactId>**spring-cloud-starter-zipkin**</artifactId>
- </dependency>

ZIPKIN – SLEUTH

- Sleuth is a tool from Spring cloud family. It is used to generate the trace id, span id and add these information to the service calls in the headers and MDC, so that It can be used by tools like Zipkin and ELK etc. to store, index and process log files.

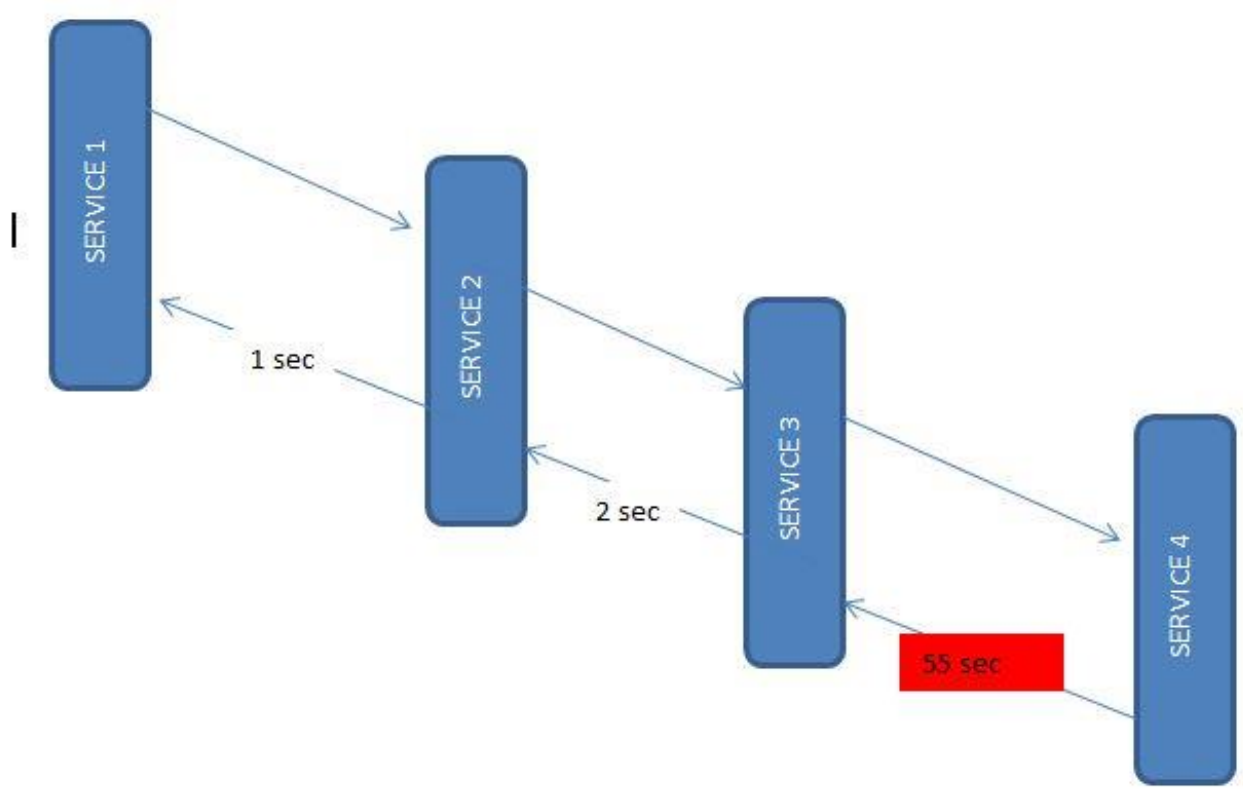
- As it is from spring cloud family, once added to the CLASSPATH, it automatically integrated to the common communication channels like –
  - requests made with the RestTemplate etc.
  - requests that pass through a Netflix Zuul microproxy
  - HTTP headers received at Spring MVC controllers
  - requests over messaging technologies like Apache Kafka or RabbitMQ etc.

ZIPKIN – SLEUTH

- Using Sleuth is very easy. We just need to add it's started pom in the spring boot project. It will add the Sleuth to project and so in its runtime.

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

ZIPKIN – SLEUTH



THANK YOU