

Objectives

- At the end of this module, you will be able to
 - Explain what Passive Service Discovery is
 - Build and Run and Spring Cloud Eureka Server
 - Build, Run, and Configure a Eureka Client

Service Discovery - Analogy

- When you sign into a chat client, what happens?
 - Client 'registers' itself with the server – server knows you are online.
 - The server provides you with a list of all the other known clients
- In essence, your client has “discovered” the other clients
 - ...and has itself been “discovered” by others



Search people...

- Anson Hoyt
- Erik St. Martin
- Hector Virgen
- Joe Sanantonio
- Kevin Crocker
- Lisa Fernandez
- Wei Teh
- Advitiya Banga
- Axel Ulrich

Service Discovery

- Microservice architectures result in large numbers of inter-service calls
 - Very challenging to configure
- How can one application easily find all of the other runtime dependencies?
 - Manual configuration – Impractical, brittle
- Service Discovery provides a single 'lookup' service.
 - Clients register themselves, discover other registrants.
 - Solutions: Eureka, Consul, Etcd, Zookeeper, SmartStack, etc.



Eureka – Service Discovery Server and Client

- Part of Spring Cloud Netflix
 - Battle tested by Netflix
- Eureka provides a 'lookup' server.
 - Generally made highly available by running multiple copies
 - Copies replicate state of registered services.
- “Client” Services register with Eureka
 - Provide metadata on host, port, health indicator URL, etc.
- Client Services send heartbeats to Eureka
 - Eureka removes services without heartbeats.

Multiple Servers Configuration

- Common Configuration Options for Eureka Server:
 - See <https://github.com/Netflix/eureka/wiki/Configuring-Eureka> for full list.

Control http port (any boot application)

```
server:  
  port: 8011  
eureka:  
  instance:  
    statusPageUrlPath: ${management.contextPath}/info  
    healthCheckUrlPath: ${management.contextPath}/health  
    hostname: localhost  
  client:  
    registerWithEureka: false  
    fetchRegistry: false  
    serviceUrl:  
      defaultZone: http://server:port/eureka/,http://server:port/eureka/
```

Comma separated list

Module Outline

- Service Discovery
- Eureka Server
- **Discovery Client**
- Service Discovery Considerations

Spring Cloud Ribbon

Understanding and Using Ribbon,
The client side load balancer

What is a Load Balancer?

▶ Traditional load balancers are server-side components

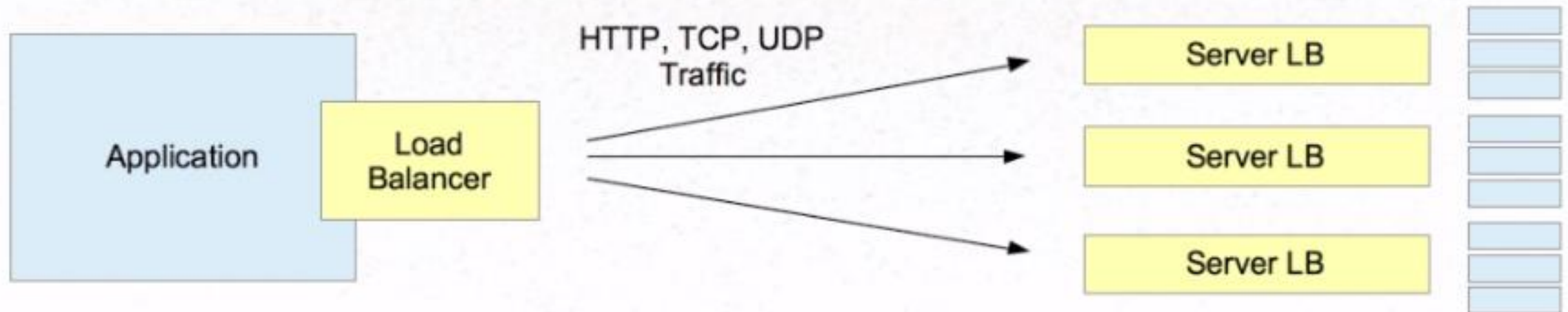
- Distribute incoming traffic among several servers
- Software (Apache, Nginx, HA Proxy) or Hardware (F5, NSX, BigIP)



Client-Side Load Balancer

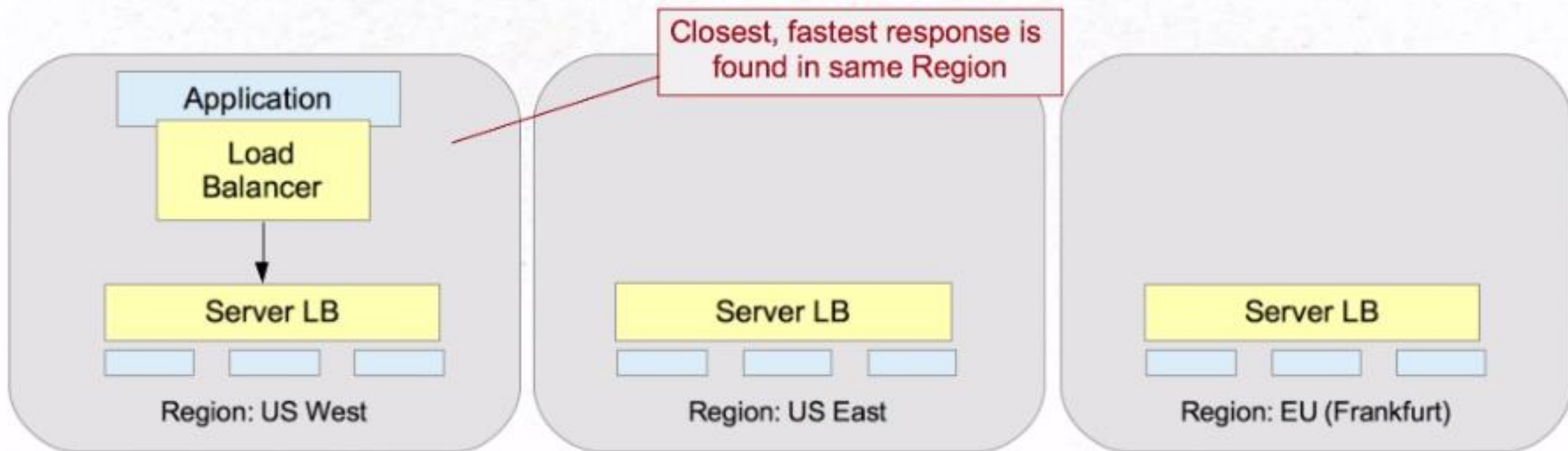
▶ Client-Side Load Balancer selects which server to call

- Based on some criteria
- Part of client software
- Server can still employ its own load balancer



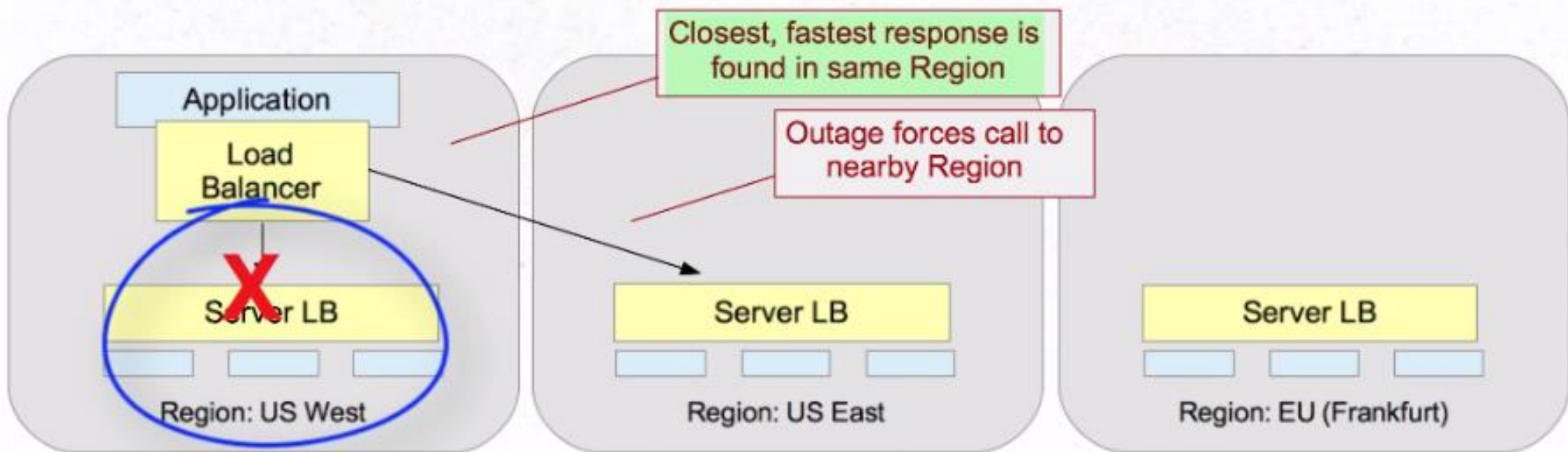
Why?

- Not all servers are the same
 - Some may be unavailable (faults)
 - Some may be slower than others (performance)
 - Some may be further away than others (regions)



Why?

- Not all servers are the same
 - Some may be unavailable (faults)
 - Some may be slower than others (performance)
 - Some may be further away than others (regions)



Module Outline

- Client Side Load Balancing
- **Spring Cloud Netflix Ribbon**

✓

Spring Cloud Netflix Ribbon

- ▶ Ribbon – Another part of the Netflix OSS family
 - Client side load balancer
 - Automatically integrates with service discovery (Eureka)
 - Built in failure resiliency (Hystrix)
 - Caching / Batching
 - Multiple protocols (HTTP, TCP, UDP)
- Spring Cloud provides an easy API Wrapper for using Ribbon.

Key Ribbon Concepts

- List of Servers
- Filtered List of Servers
- Load Balancer
- Ping

List of Servers

- Determines what the list of possible servers are (for a given service (client))
 - Static – Populated via configuration
 - Dynamic – Populated via Service Discovery (Eureka)
- Spring Cloud default – Use Eureka when present on the classpath.

Example of "Static" server lists

application.yml

```
stores:
  ribbon:
    listOfServers: store1.com,store2.com
products:
  ribbon:
    listOfServers: productServer1.com, productServer2.com
```

"stores" and "products" -
Examples of client-ids

Filtered List of Servers

- Criteria by which you wish to limit the total list
- Spring Cloud default – Filter servers in the same *zone*

Ping

- Used to test if the server is up or down
- Spring Cloud default – delegate to Eureka to determine if server is up or down

Load Balancer

- The Load Balancer is the actual component that routes the calls to the servers in the filtered list
- Several strategies available, but they usually defer to a Rule component to make the actual decisions
- Spring Cloud's Default: ZoneAwareLoadBalancer

Rule

- The Rule is the single module of intelligence that makes the decisions on whether to call or not.
- Spring Cloud's Default: `ZoneAvoidanceRule`

Using Ribbon with Spring Cloud — part 1

- Use the Spring Cloud Starter parent as a Parent POM:

```
<parent>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-parent</artifactId>
  <version>Angel.SR4</version>
</parent>
```

- ...OR use a Dependency management section:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-parent</artifactId>
      <version>Angel.SR4</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

...exactly the same options
as a spring cloud config client
or a spring cloud eureka client.

Using Ribbon with Spring Cloud – part 2

- Include dependency:

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

Using Ribbon with Spring Cloud – part 3

- Low-level technique:
 - Access LoadBalancer, use directly:

```
public class MyClass {  
    @Autowired LoadBalancerClient loadBalancer;  
  
    public void doStuff() {  
        ServiceInstance instance = loadBalancer.choose("subject");  
        URI subjectUri = URI.create(String.format("http://%s:%s",  
            instance.getHost(), instance.getPort()));  
        // ... do something with the URI  
    }  
}
```

"subject" - An example of a "client-id"

Instance selected
by the
load balancer

API Reference

- ▶ Previous example used Ribbon API directly
 - Not desirable – couples code to Ribbon
- Upcoming examples will show declarative approach
 - Feign, Hystrix.

Customizing

- Previously we described the defaults. What if you want to change them?
- Declare a separate config with replacement bean.

```
@Configuration
@RibbonClient(name="subject", configuration=SubjectConfig.class)
public class MainConfig {
}
```

```
@Configuration
public class SubjectConfig {
    @Bean
    public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
    }
}
```

Replaces the "Ping" strategy
Employed when calling "subject" clients

Note: Do NOT component scan
SubjectConfig!

Summary

Client-Side Load Balancing augments regular load balancing by allowing the client to select a server based on some criteria.

Spring Cloud Ribbon is an easy-to-use implementation of client side load balancing.

Spring Cloud Feign

Declarative REST Client

Objectives

- At the end of this module, you will be able to
 - ▶ Call REST services using the Feign libraries
 - Understand how Feign, Ribbon, and Eureka collaborate

Module Outline

- What is Feign
- How to use Feign

Feign

- What is it?
 - Declarative REST client, from Netflix
 - Allows you to write calls to REST services with no implementation code
 - Alternative to RestTemplate (even easier!)
 - Spring Cloud provides easy wrapper for using Feign

Spring REST Template

- Spring's Rest Template provides very easy way to call REST services

```
RestTemplate template = new RestTemplate();  
String url = "http://inventoryService/{0}";  
Sku sku = template.getForObject(url, Sku.class, 4724352);
```

Instantiate
(or dependency inject)

Provide target URL
(note the placeholder)

Call the URL, provide expected class,
Provide value for placeholder.
Template takes care of all HTTP
and type conversion!

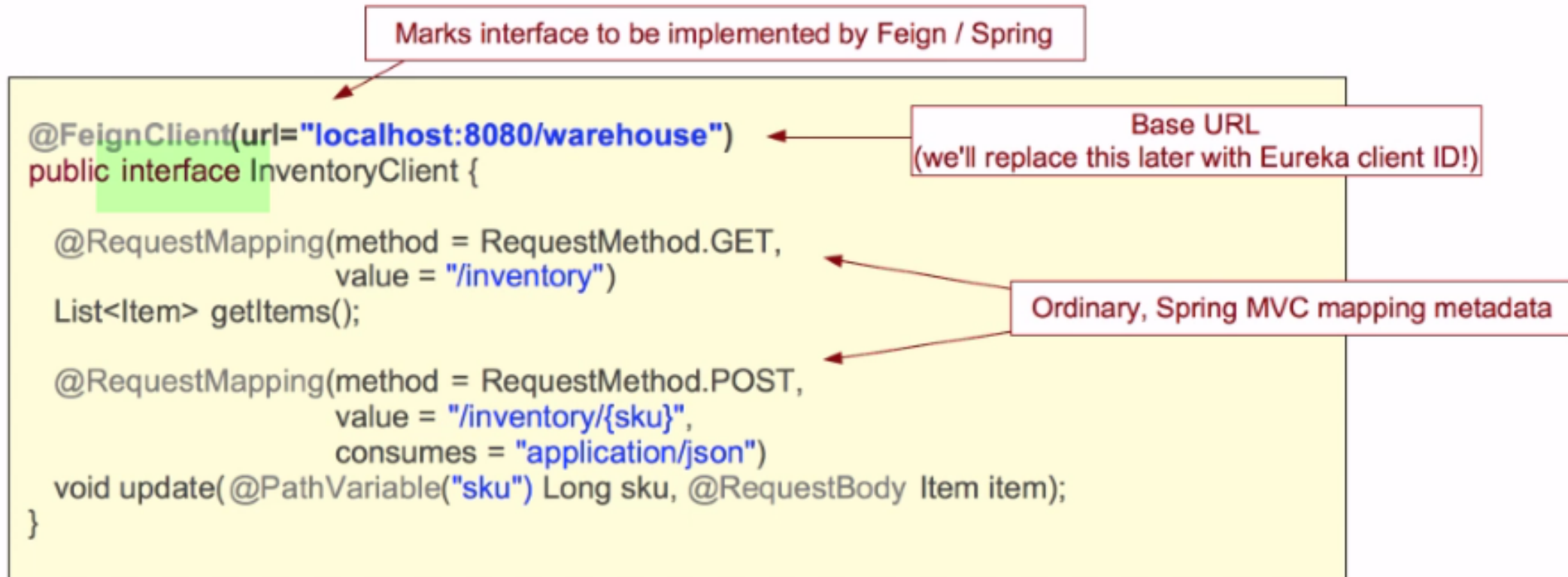
- Still, this code must be
 - 1) Written
 - 2) Unit-tested with mocks / stubs.

Feign Alternative – Declarative Web Service Clients

- How does it work?
 - Define *interfaces* for your REST client code
 - Annotate interface with Feign annotation
 - Annotate methods with Spring MVC annotations
 - Other implementations like JAX/RS pluggable
- Spring Cloud will implement it at run-time
 - Scans for interfaces
 - Automatically implements code to call REST service and process response

Feign Interface

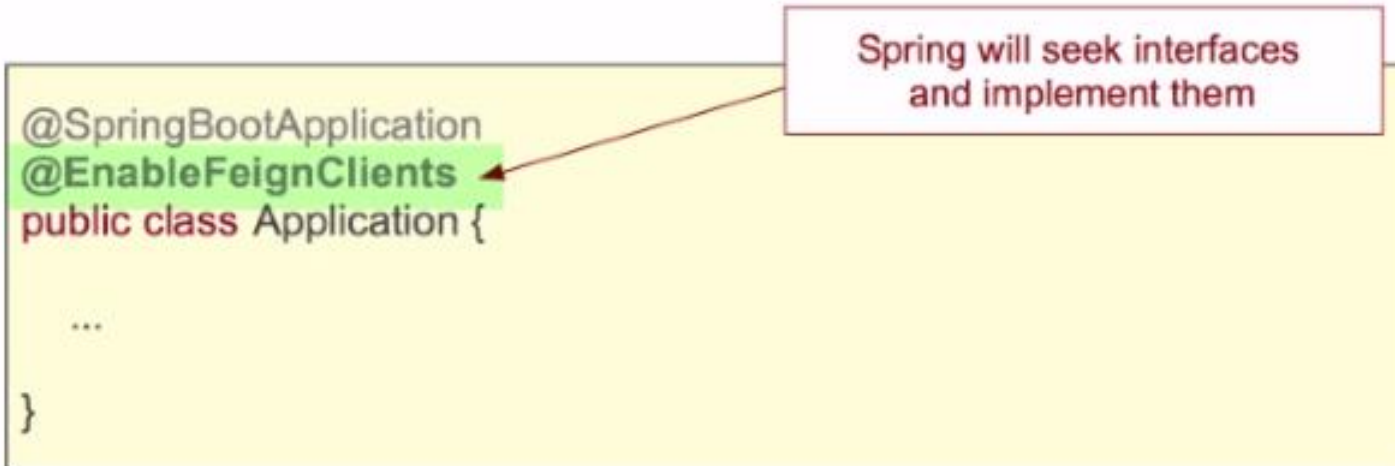
- Create an *Interface*, not a Class:



Note: No extra dependencies are needed for Feign when using Spring Cloud.

Runtime Implementations

- ▶ Spring scans for @FeignClients
 - Provides implementations at runtime



- That's it!
 - Implementations provided by Spring / Feign!

What does @EnableFeignClients do?

- Before startup

InventoryClient
(Java interface)

- After startup

InventoryClient
(Java interface)

↑ *implements*

Spring-Implemented
Proxy

@EnableFeignClients

You can @Autowire an InventoryClient wherever one is needed

Ribbon and Eureka

Where do they fit in?

- The previous example - hard-coded URL:

```
@FeignClient(url="localhost:8080/warehouse")
```

- ...use a Eureka “Client ID” instead:

```
@FeignClient("warehouse")
```

- Ribbon is automatically enabled
 - Eureka gives our application all “Clients” that match the given Client ID
 - Ribbon automatically applies load balancing
 - Feign handles the code.

Runtime Dependency

- Feign starter required at runtime:
...but not compile time

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


Spring Cloud Hystrix

Understanding and Applying
Client Side Circuit Breakers

