Spring Cloud and Netflix oss

Agenda

- Intro Spring framework
- Intro Spring Boot
- Intro Spring Cloud
- Demo time

Spring framework -> How it started

- Born out of frustration with old releases of JEE
  - Developers working with Glassfish, Websphere, WebLogic, Oracle Application Server,..
  - Steep learning curve
  - Theoretically ideal -> practically hellish

JEE in practice

- One Standard to rule them all sounds great but ...
  - Creating standards is a slow process
  - Vested interests
  - Hidden agenda's
  - Victims -> the developers -> customers

A JEE Architect Rod Johnson comes to the rescue

- Wrote a book about problems and possible solutions in JEE
  - Enormous response
  - Start of a community
  - A community deeply rooted in the field
  - Start of the Spring software

Essential Characteristics Spring Software

- Some Guiding principles
  - Software should work on every container (JEE)
  - Easy for the developer
  - Practical/Sensible defaults

Birth of SpringContainer

- ApplicationContext a.k.a. SpringContainer supplies the springgoodies
  - Pojo-fied
  - Dependency Injection
  - AOP
  - Portable Service Abstractions

Spring is Great

- One downside though..
  - Spring can do everything but
  - You had to configure it first with XML
  - lots of XML :-) :-) after some time
  - lots of @Annotations :-) after some time
  - a few powerAnnotations :-) :-)

Spring Boot to the rescue

- Power Annotations are part of Spring Boot
  - Opineated - out of the box - solutions
  - Minimal configuration
  - Arrangement of highly popular software
  - Pick and choose via start.spring.io

Spring Boot and Micro-services

- Easy to configure small task focused applications with Boot
  - Boot provisions a starter application
  - Ideal for microservices

What are micro-services?

- No monoliths
- Smaller units of a larger system
- Runs in its own process
- Single Responsibility Principle
- The unix way

Spring Cloud

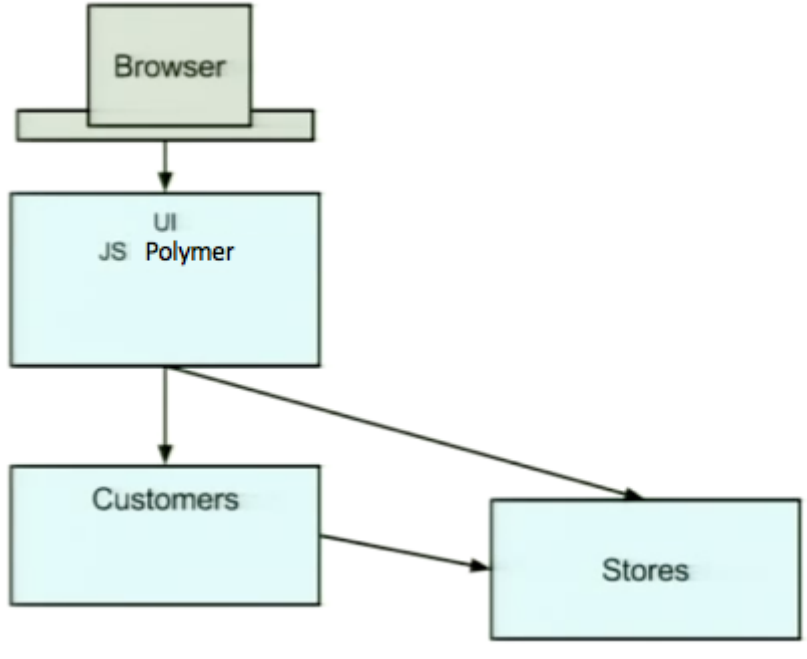
- Combining these micro-services is not easy
  - A lot of recurring problems to solve
  - configuration
  - discovering
  - load balancing etc
- Enter Spring Cloud -> Spring Boot for cloud products

Spring Cloud and Netflix oss

- Netflix oss is a set of cloudproducts donated by netflix to community
- Netflix oss bootified
- Netflix oss first because of existing opensource community
- Other cloud products are also incorporated

A small microservice application

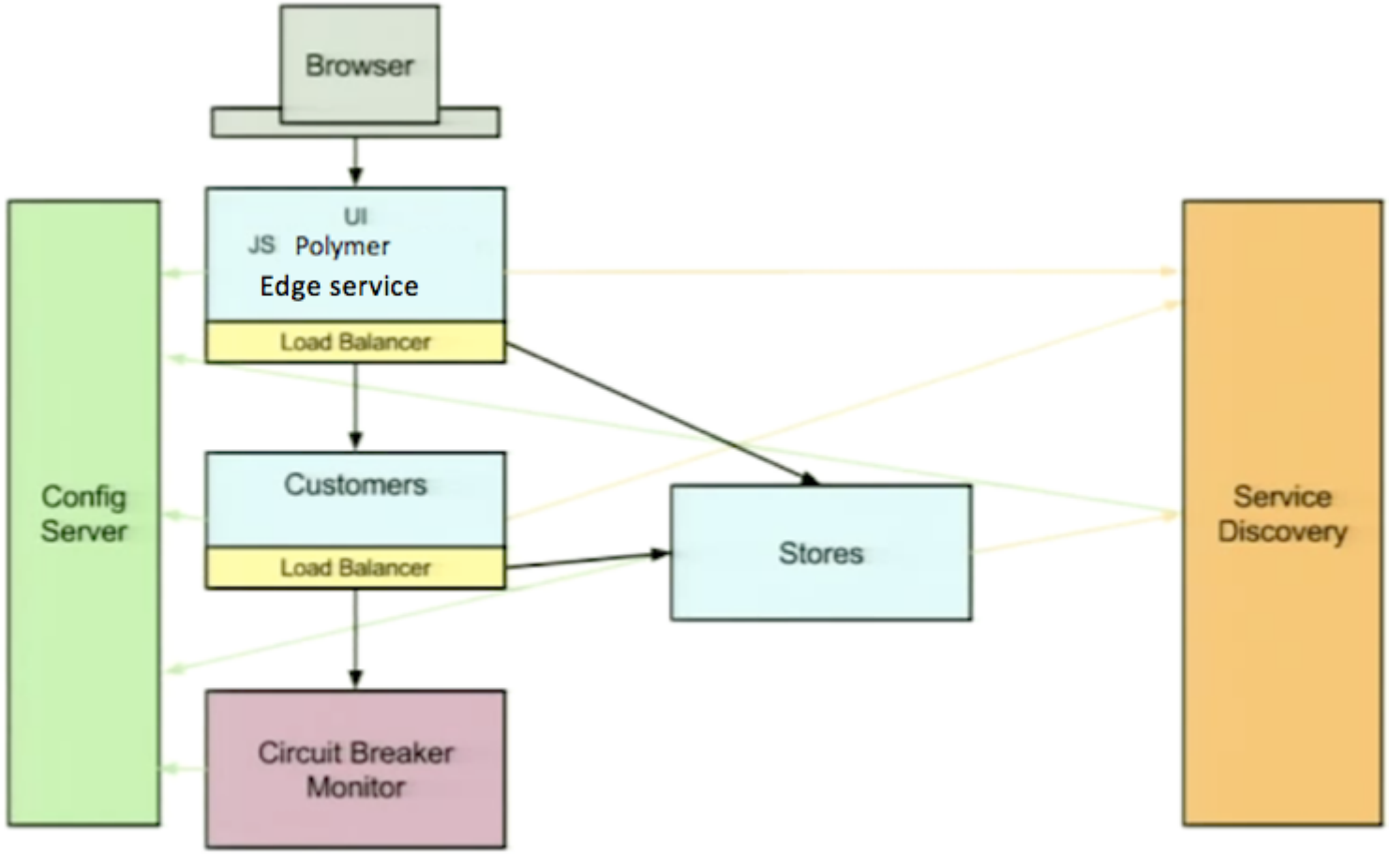
- three microservices



A distributed application leads to a lot of boilerplate code

- distributed and versioned configuration data
- service registration and discovery
- routing
- service to service calls
- load balancing
- circuit breaker
- asynchronous
- distributed messaging

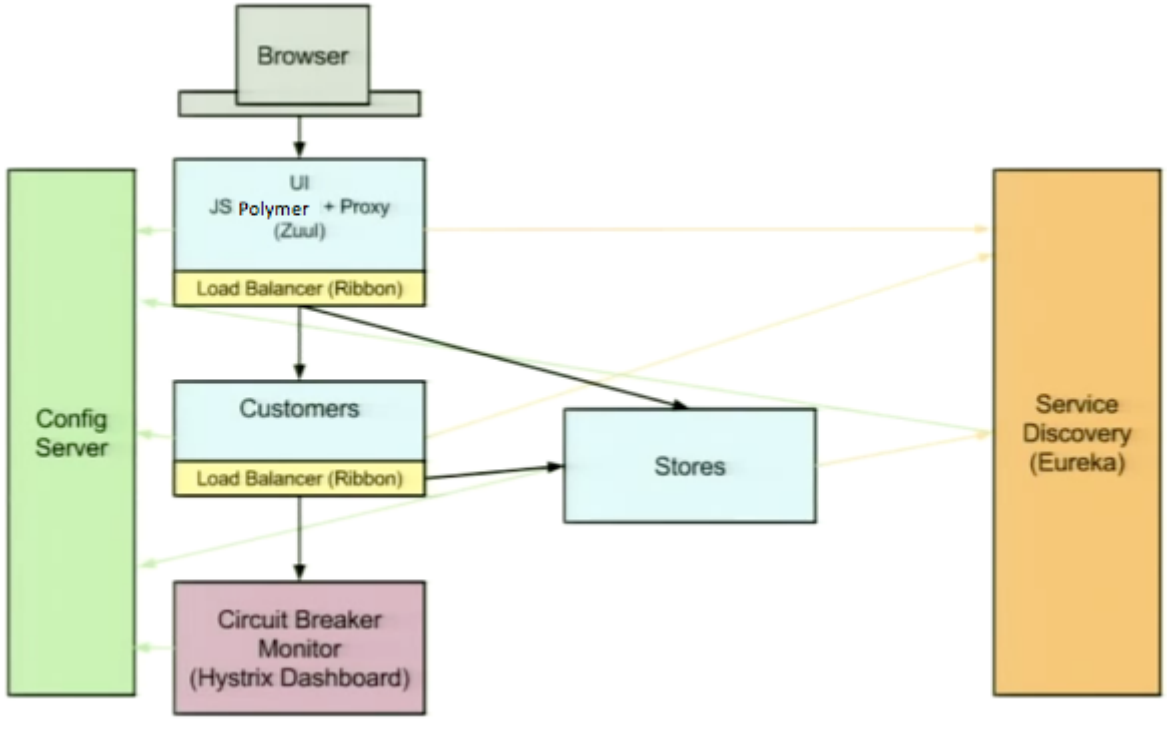
Where do we need supporting functionality?

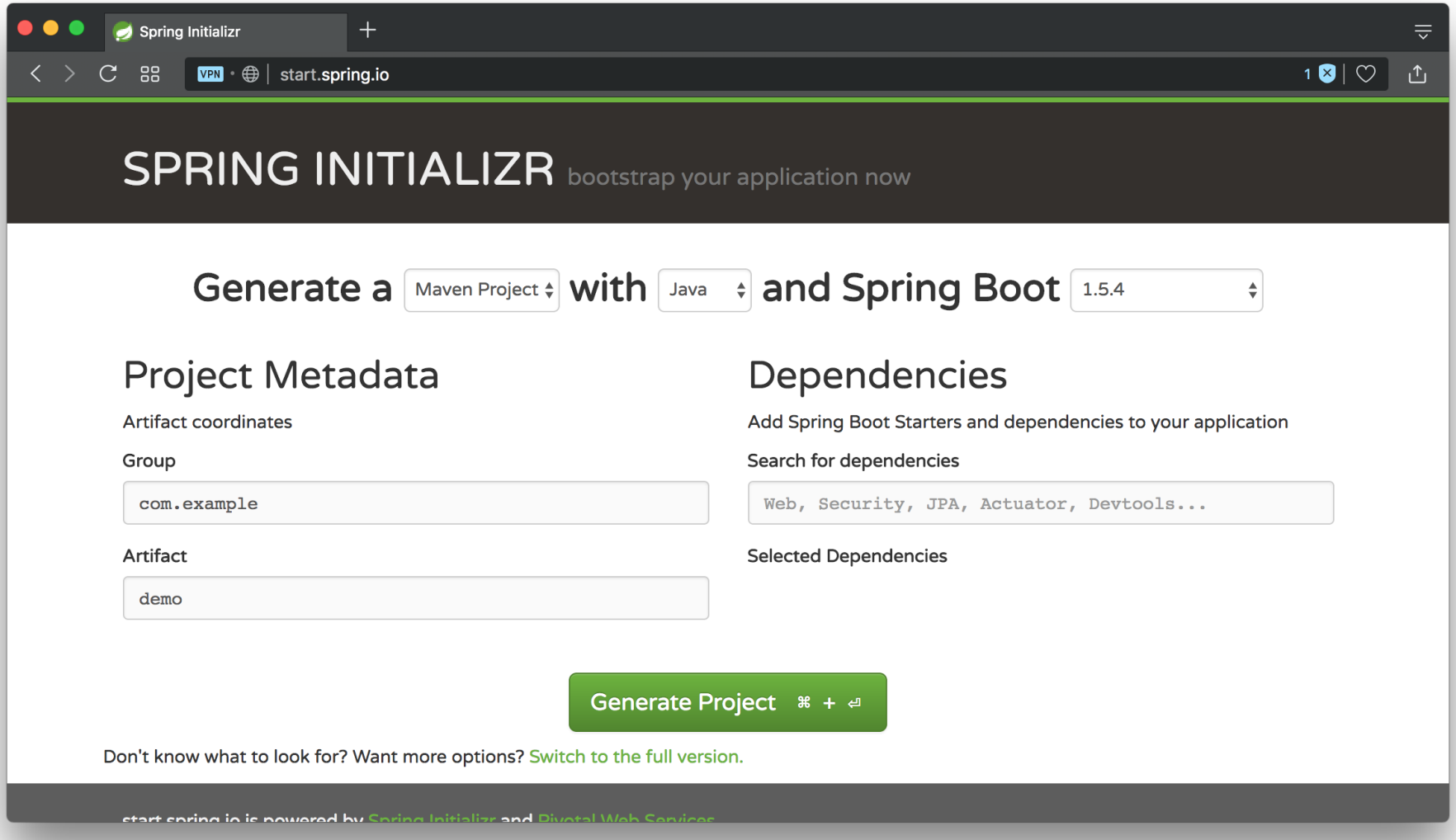


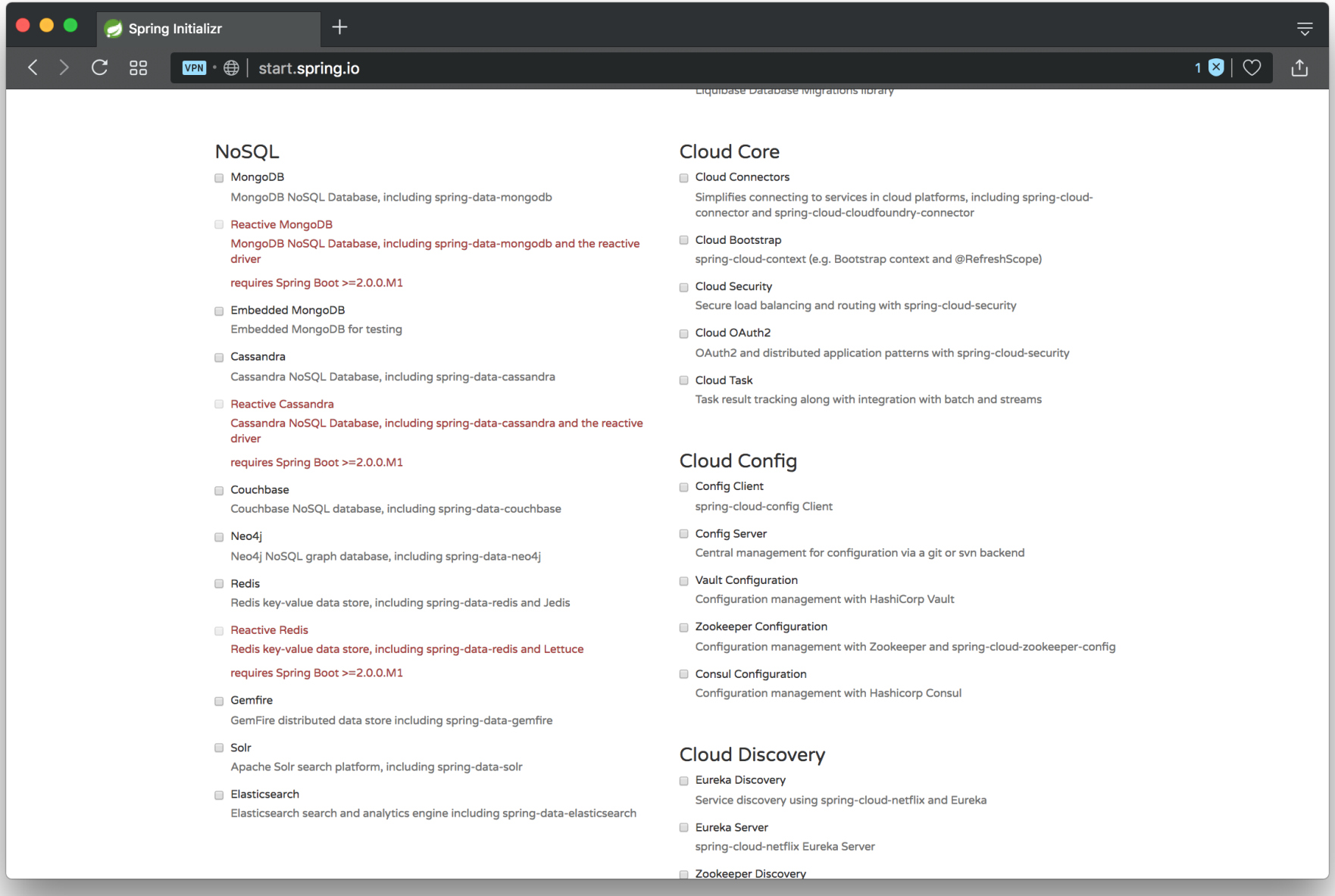
Netflix oss functionality

- Eureka
- Hysterix & Turbine
- Ribbon
- Feign
- Zuul
- Archaius
- Curator
- Asgaard

Putting Netflix oss to use in application




Spring Initializr
start.spring.io
SPRING INITIALIZR bootstrap your application now
Generate a Maven Project with Java and Spring Boot 1.5.4
Project Metadata
Artifact coordinates
Group
com.example
Artifact
demo
Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies
Web, Security, JPA, Actuator, Devtools...
Selected Dependencies
Generate Project ⌘ + ⏎
Don't know what to look for? Want more options? Switch to the full version.
start.spring.io is powered by Spring Initializr and Pivotal Web Services


Spring Initializr
start.spring.io
Liquibase Database Migrations library
NoSQL
MongoDB
MongoDB NoSQL Database, including spring-data-mongodb
Reactive MongoDB
MongoDB NoSQL Database, including spring-data-mongodb and the reactive driver
requires Spring Boot >=2.0.0.M1
Embedded MongoDB
Embedded MongoDB for testing
Cassandra
Cassandra NoSQL Database, including spring-data-cassandra
Reactive Cassandra
Cassandra NoSQL Database, including spring-data-cassandra and the reactive driver
requires Spring Boot >=2.0.0.M1
Couchbase
Couchbase NoSQL database, including spring-data-couchbase
Neo4j
Neo4j NoSQL graph database, including spring-data-neo4j
Redis
Redis key-value data store, including spring-data-redis and Jedis
Reactive Redis
Redis key-value data store, including spring-data-redis and Lettuce
requires Spring Boot >=2.0.0.M1
Gemfire
GemFire distributed data store including spring-data-gemfire
Solr
Apache Solr search platform, including spring-data-solr
Elasticsearch
Elasticsearch search and analytics engine including spring-data-elasticsearch
Cloud Core
Cloud Connectors
Simplifies connecting to services in cloud platforms, including spring-cloud-connector and spring-cloud-cloudfoundry-connector
Cloud Bootstrap
spring-cloud-context (e.g. Bootstrap context and @RefreshScope)
Cloud Security
Secure load balancing and routing with spring-cloud-security
Cloud OAuth2
OAuth2 and distributed application patterns with spring-cloud-security
Cloud Task
Task result tracking along with integration with batch and streams
Cloud Config
Config Client
spring-cloud-config Client
Config Server
Central management for configuration via a git or svn backend
Vault Configuration
Configuration management with HashiCorp Vault
Zookeeper Configuration
Configuration management with Zookeeper and spring-cloud-zookeeper-config
Consul Configuration
Configuration management with Hashicorp Consul
Cloud Discovery
Eureka Discovery
Service discovery using spring-cloud-netflix and Eureka
Eureka Server
spring-cloud-netflix Eureka Server
Zookeeper Discovery

A lot to read at:

spring documentation

Spring Cloud Configuration Server

- Git implementation
- Versioned
- Rollback-able
- Configuration client auto-configured via starter

Different kind of property files

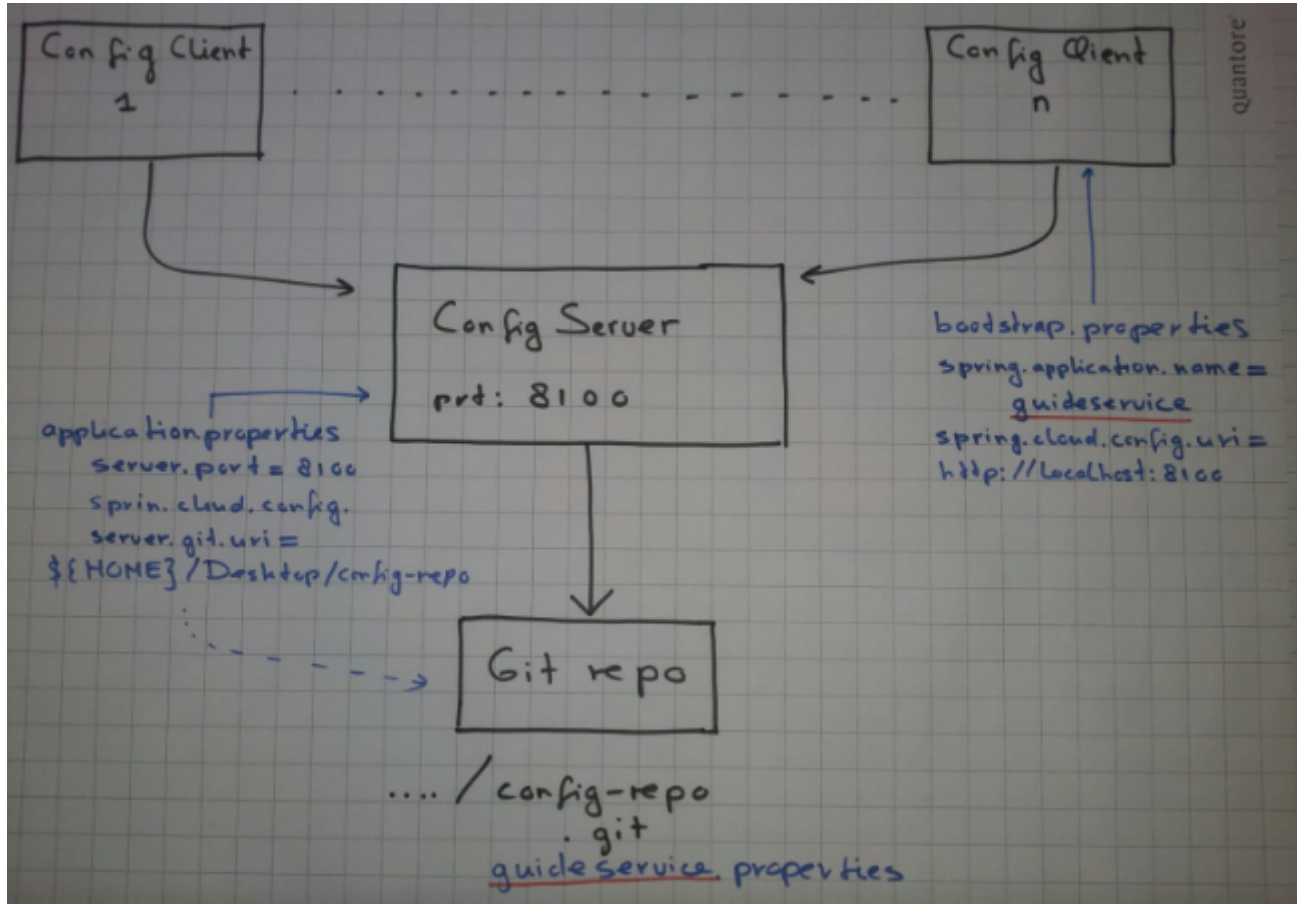
- Supports applications <appname>.properties
- Supports environments <appname>-<envname>.yml
- Default environment application.properties applies to all applications and environments

Change values in the environment

- Bootstrap Environment from server
- POST to /env to change Environment
- @RefreshScope for atomic changes to beans via Spring lifecycle
- POST to /refresh
- POST to /restart

Demo Config Server

- What to build?



Creating ConfigServer

- @EnableConfigServer does the trick

```
@EnableConfigServer
@SpringBootApplication
public class ConfigurationServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigurationServerApplication.clas
    }
}
```

Prerequisite @EnableConfigServer

- When necessary dependencies are present on classpath
  - @EnableConfigServer will configure the server

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

Configure git as repository

- in src/main/resources/application.properties

```
server.port=8100
# port to service configuration data requests
spring.cloud.config.server.git.uri=${HOME}/config-repo
# config-repo contains the git repo (.git directory)
```

- With git we can version controlled, service, configuration requests

The Configure Client

- A normal Springboot application
- It must be configured to get it's configuration from ConfigServer
  - Use bootstrap.properties to point Config client to ConfigServer
  - bootstrap.properties is read before application.properties

Config data: answerToEverything

```
@RefreshScope
@RestController
public class RESTController {
 @Value("${answerToEverything:0}")
 private String answerToEverything;

 @RequestMapping("/answer")
 String getExample() {
   return answerToEverything;
 }
}
```

- @Value -> get settings from envirionment
- ${} -> Spring expression syntax

Picking up changes

- When data on ConfigServer changes
  - change can be picked up by using RefreshScoped bean

```
@RefreshScope
@RestController
public class RESTController {...}
```

- POST on the refresh endpoint /refresh of client service
- client is forced to read the envirionment again
- only works when actuator dependencies are included

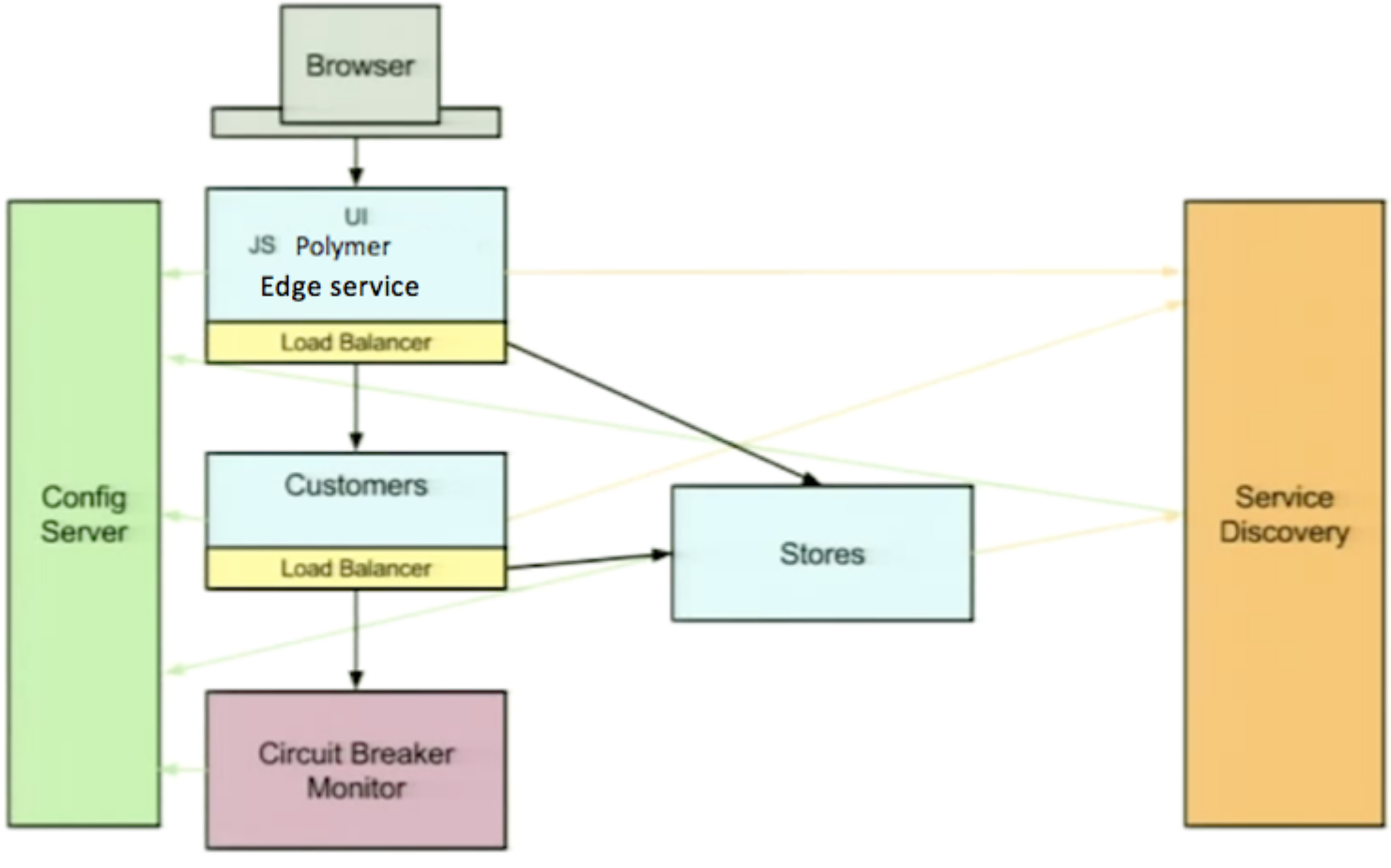
Necessary dependencies

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- For refresh -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
</dependencies>
```

Discovery -> Eureka

- Service Registration Server
- Highly available
- Multi availability zone, region aware

Focus on EurekaServer



Creating EurekaServer

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {

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

- That's it, apart from necessary dependencies and configuaration

Dependencies and config

- from pom.xml

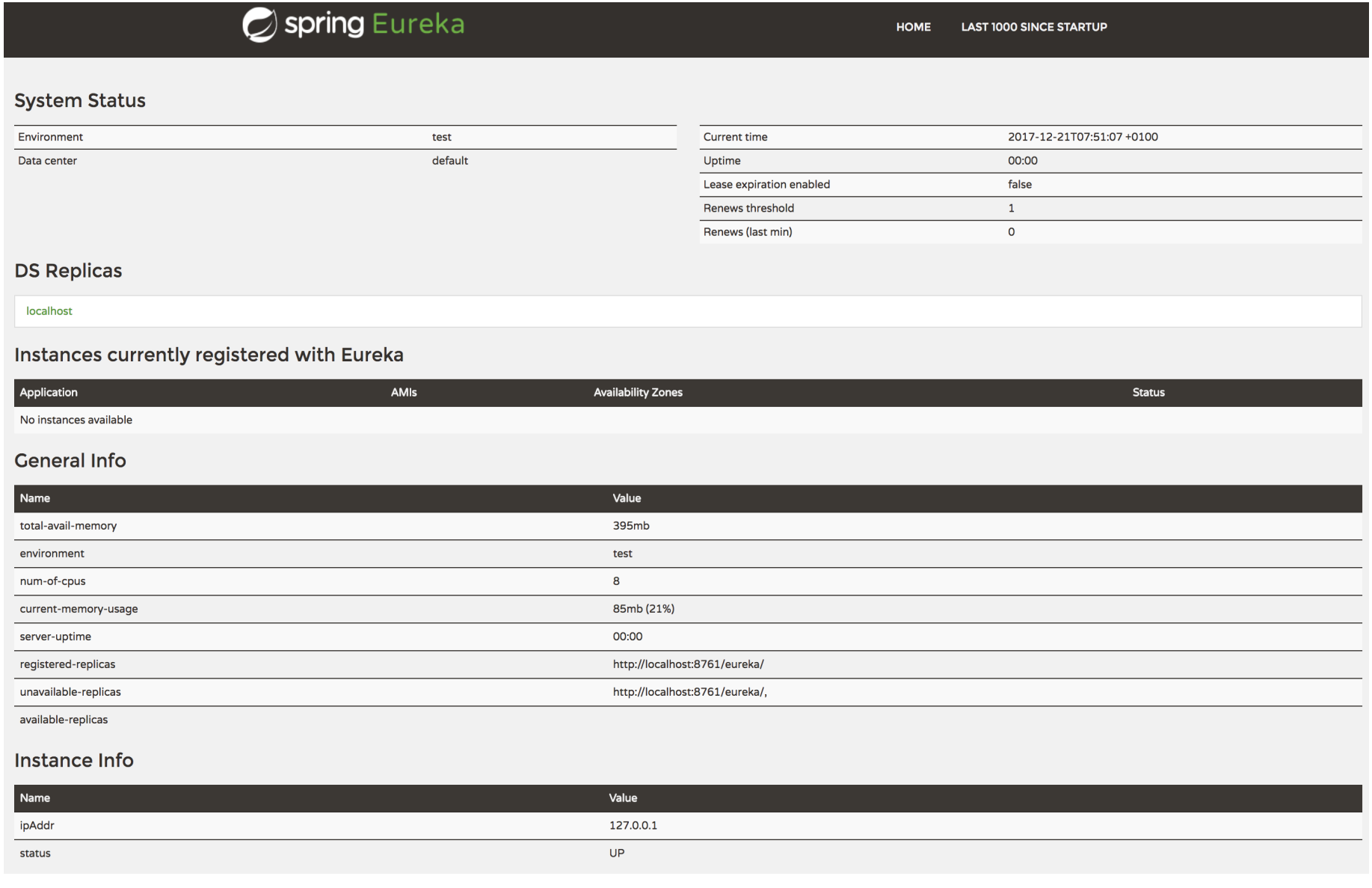
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

- application.properties

```
server.port=8010
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

- register-with-eureka -> do not register this eurekaservice to eureka

Eureka home page



Eureka client

- Register service instances with Eureka Server
- @EnableEurekaClient auto registers instance in server
- Eureka Server
- Eureka Client

@EnableDiscoveryClient

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

- EurekaServiceApplication -> provides a service
  - compare role to StoreService, or CustomerService
- Let Spring Cloud enable/configure "eurekaclient"

DiscoveryClient

```
@RestController
public class RESTController {
 @Autowired
 DiscoveryClient client;
 @RequestMapping("/")
 public String retrieveServiceInfo() {
  ServiceInstance instance = client.getLocalServiceInstance();
  return "Service info. ID: " + instance.getServiceId()+
  " host: " + instance.getHost() + " port: " + instance.getPort()
 }
}
```

- DiscoveryClient communicates with the DiscoveryServer
  - (here) eureka (Netflix)
- DiscoveryClient registers this service in EurekaServer
- Clients lookup this service address in EurekaServer
  - This service also uses Eureka for needed services

Explanation Code

```
@RequestMapping("/")
public String retrieveServiceInfo() {
 ServiceInstance instance = client.getLocalServiceInstance();
 return "Service info. ID: " + instance.getServiceId()+
 " host: " + instance.getHost() + " port: " + instance.getPort();
}
```

- This service, instance, fetches via the discoveryClient, client, eureka data
- The eureka data represents the registrationdata in eureka of this service

EurekaService Configuration

```
spring.application.name=serviceinfo

# Necessary for Docker as it doesn't have DNS entries
eureka.instance.preferIpAddress=true
eureka.client.serviceUrl.defaultZone=http://localhost:8010/eureka
server.port=8053
```

- start 2 instances of serviceinfo service
- 1 instance on port 8053 and one on port 8054

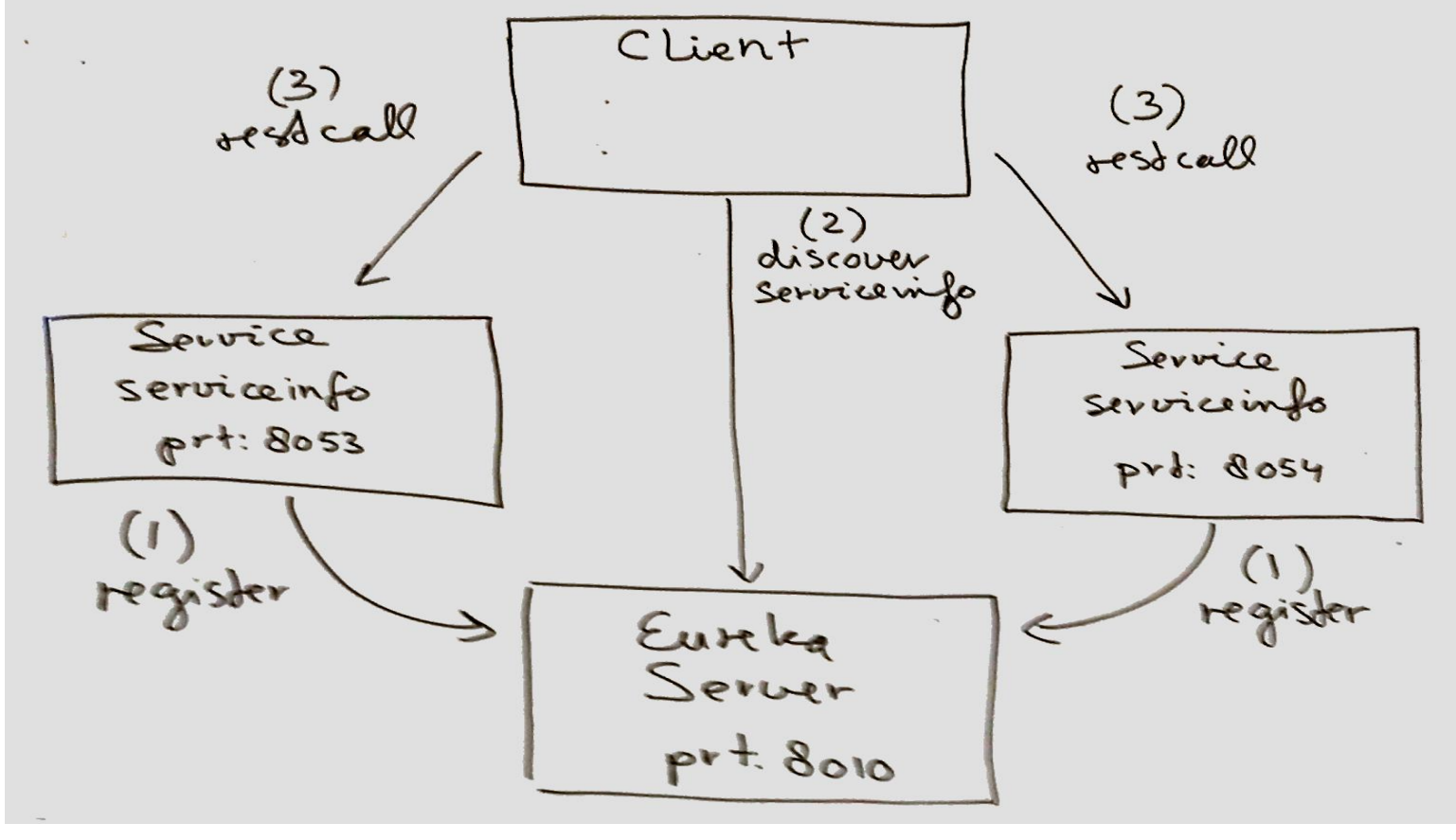
Eureka Server Data

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SERVICEINFO	n/a (2)	(2)	UP (2) - Joriss-MacBook-Pro.local:serviceinfo:8054 , Joriss-MacBook-Pro.local:serviceinfo:8053

A Client using DiscoveryServer

- What to achieve:

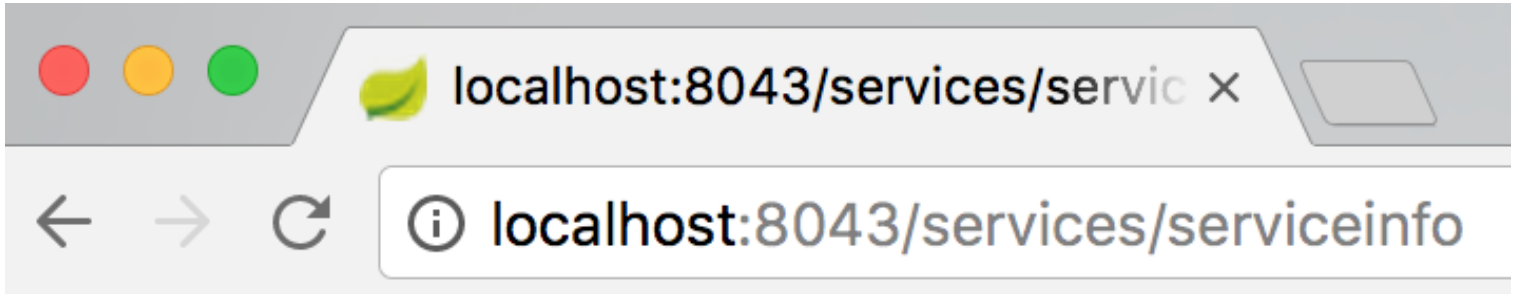


The client code

```
@RestController
public class RESTController {
 @Autowired
 private DiscoveryClient discoveryClient;
 // {servicename} name of EurekaService we want to use
 @RequestMapping("/services/{name}")
 public List<ServiceInstance> services(@PathVariable String name)
   return discoveryClient.getInstances(name);
 }
}
```

- Use DiscoveryClient
  - Look up service in eureka via servicename
- Confusing:
  - Service called returns it's runtime eurekaServer data

The client request



The Service Response


```
[
 {
    "host": "127.0.0.1",
    "port": 8053,
    "uri": "http://127.0.0.1:8053",
    ... removed some attributes
    "serviceId": "SERVICEINFO"
 },
 {
    "host": "127.0.0.1",
    "port": 8054,
    "uri": "http://127.0.0.1:8054",
    ... removed some attributes
    "serviceId": "SERVICEINFO"
 }
]
```

Zooming in on some attributes


```
"instanceInfo": {
    "instanceId": "Joriss-MacBook-Pro.local:serviceinfo:8053",
    "app": "SERVICEINFO",
    ... removed a lot of attributes
    "homePageUrl": "http://127.0.0.1:8053/",
    "statusPageUrl": "http://127.0.0.1:8053/info",
    "healthCheckUrl": "http://127.0.0.1:8053/health",
    "vipAddress": "serviceinfo",
    "secureVipAddress": "serviceinfo",
    "dataCenterInfo": {..},
    "hostName": "127.0.0.1",
    "status": "UP",
    "leaseInfo": {..},
    "isCoordinatingDiscoveryServer": false,
}
```

Ribbon

- Client side load balancer
- Protocols: http, tcp, udp
  - Pluggable transport
- Pluggable load balancing algorithms
  - Round robin, best available, random, response time based

@LoadBalanced

- configure RestTemplate to use a LoadBalancerClient

```
@SpringBootApplication
public class EurekaClientRibbonApplication {

 public static void main(String[] args) {
    SpringApplication.run(EurekaClientRibbonApplication.class, ar
 }
 @Bean
 @LoadBalanced
 public RestTemplate restTemplate() {
    return new RestTemplate();
 }
}
```

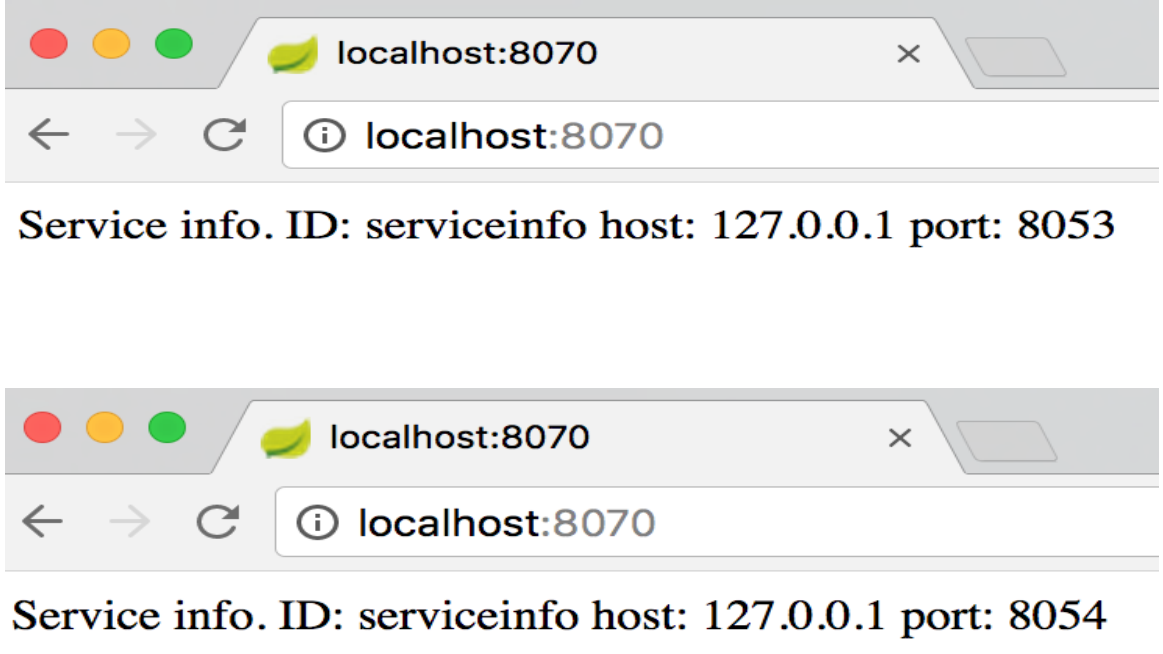
Call the serviceinfo service

- Call the serviceinfo service, 2 instances are up

```
@RestController
public class RESTController {
 @Autowired
 private RestTemplate restTemplate;

 @RequestMapping("/")
 public String test() {
    return restTemplate.getForEntity("http://serviceinfo/", Strin
 }
}
```

Consecutive Calls



- Remark: consecutive calls are routed to different ports
  - algorithm: round robin

Feign

- Declarative web service client definition
- Annotate an interface
- Highly customizable
- Encoder/Decoders
- Annotation processors(Feign, JAX-RS)
- Logging
- Supports Ribbon and therefore Eureka

@EnableFeignClients

```
@EnableFeignClients
@EnableDiscoveryClient
@SpringBootApplication
public class EurekaClientFeignApplication {

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

- Don't write rest clients -> generate runtime

Autogenration RestClient

```
@FeignClient("serviceinfo")
@RestController
public interface RESTController { //Remark: interface

  @RequestMapping(value = "/", method = RequestMethod.GET)
  String retrieveServiceInfo();
}
```

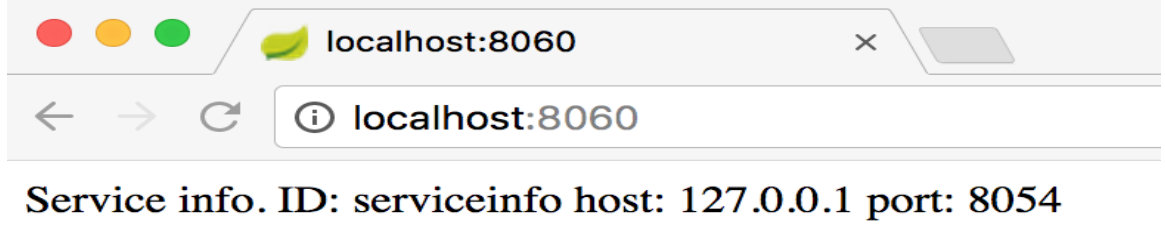
- compare with restEndpoint service called

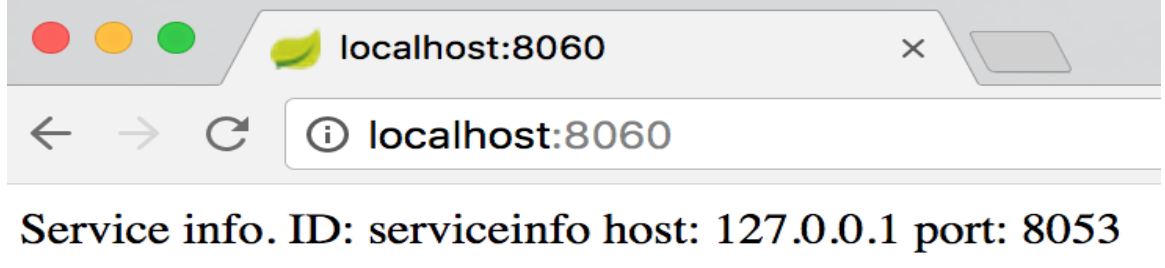
```
@RequestMapping("/")
public String retrieveServiceInfo() {
  //details
}
```

Dependencies

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

Consecutive Calls





- Remark: consecutive calls are routed to different ports
  - algorithm: round robin

Circuit Breaker Hystrix

- Latency and fault tolerance
- Isolates access to other services
- Stops cascading failures
- Enables resilience
- Circuit breaker pattern
- Dashboard

@EnableCircuitBreaker

```
@EnableCircuitBreaker
@SpringBootApplication
public class EurekaClientHystrixApplication {
 public static void main(String[] args) {
  SpringApplication.run(EurekaClientHystrixApplication.class, arg
 }
 @Bean
 @LoadBalanced
 public RestTemplate restTemplate() {
   return new RestTemplate();
 }
}
```

2 Endpoints

```
@RestController
public class RESTController {
 @Autowired
 private CommunicationService communicationService;
 @RequestMapping("/")
 public String fallback() {
   return communicationService.fallback();
 }
 @RequestMapping("/50percent")
 public String fiftyPercent() {
   return communicationService.fiftyPercent();
 }
}
```

- endpoint / all requests have delay of 2 seconds
  - represents bad connection
- endpoint /50percent every second request is 2s delayed

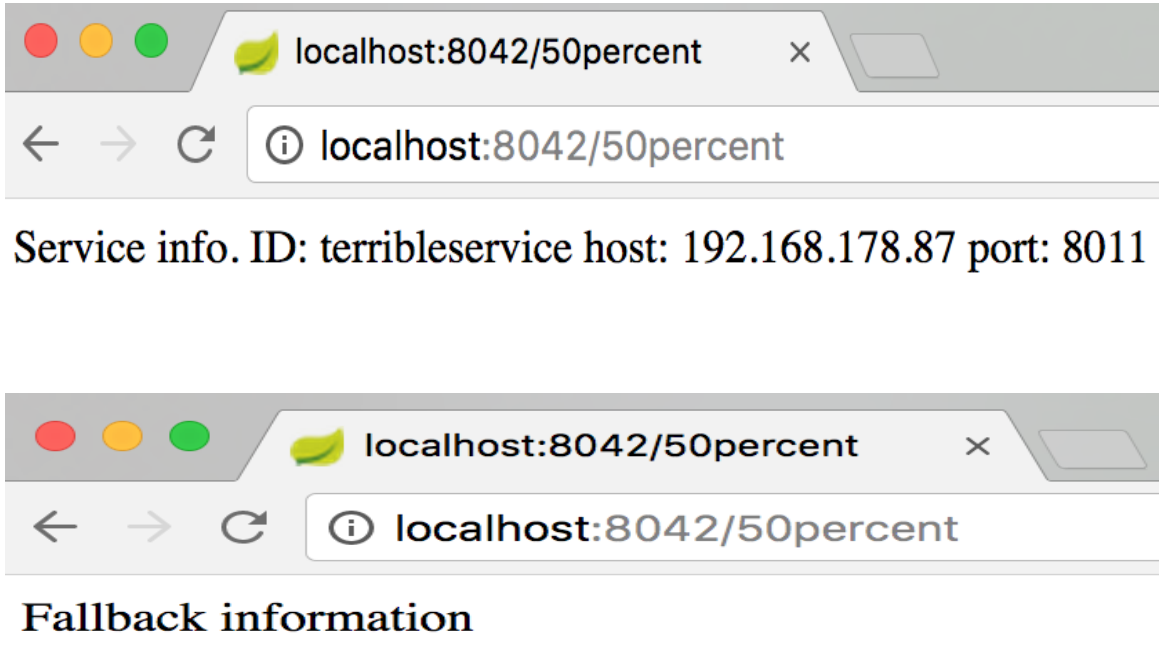
Declarative Hysterix

- Programmatic access is not easy
- @HystrixCommand to the rescue
- @EnableHystrix via starter pom
- Wires up spring aop aspect

@HystrixCommand

```
@Service
public class CommunicationService {
 @Autowired
 private RestTemplate restTemplate;
 public String fallbackText() {
    return "Fallback information";
 }
 @HystrixCommand(fallbackMethod = "fallbackText")
 public String fiftyPercent() {
  return restTemplate
          .getForEntity("http://terribleservice/50percentdelay",
                        String.class).getBody();
 }
}
```

Consecutive Calls



- Remark: hysterix becomes active the moment the request takes to long