Introduction to Spring Cloud

Microservices Approach

LXFT LISTED NYSE

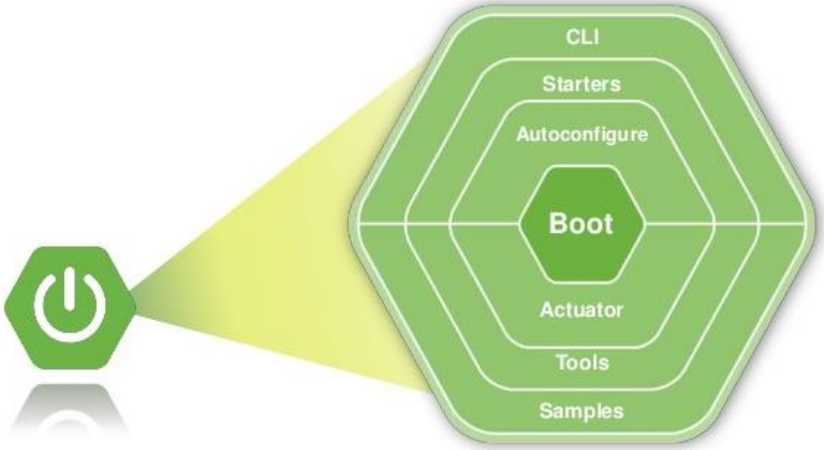
Prepared by Orkhan Gasimov
Feb. 2017

LUXOFT

Spring Cloud Netflix

Spring Cloud Netflix

- Spring Cloud applications are built with Spring Boot.
  - With Spring Boot you take advantage of the basic default behavior to get started quickly, and configure or extend a custom solution when you need to.

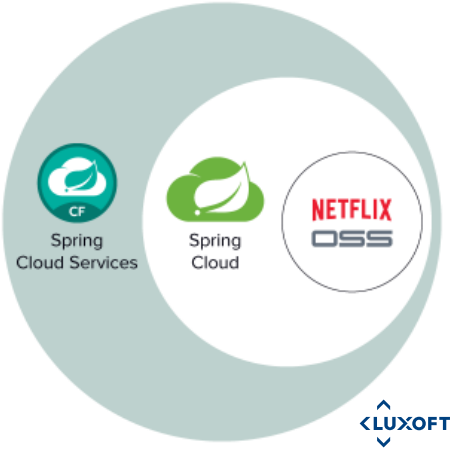


Spring Cloud Netflix

- Netflix is an internet media streaming company. Their service is a huge distributed system with more than billion of client requests a day.
  - During their research and development, they built a very productive microservices toolkit which was open sourced.
  - Netflix open-source software (OSS) was picked up by Spring Cloud to deliver a good microservices framework.

Spring Cloud Netflix

- Spring Cloud Netflix is the result of integration of Spring Framework with Netflix OSS components.
- Netflix OSS component suite consists of such components like:
  - service discovery client and server, circuit breaker, extended REST client, client-side load balancer and etc.



Microservices

Microservices

- An application design usually ends up with a set of blocks that application is decomposed to.
  - In object-oriented design, these blocks are objects that form our application.
  - In microservices approach they would be services (microservices).
- Microservices are physical services that are separately deployed standalone applications.

LUXOFT

Microservices

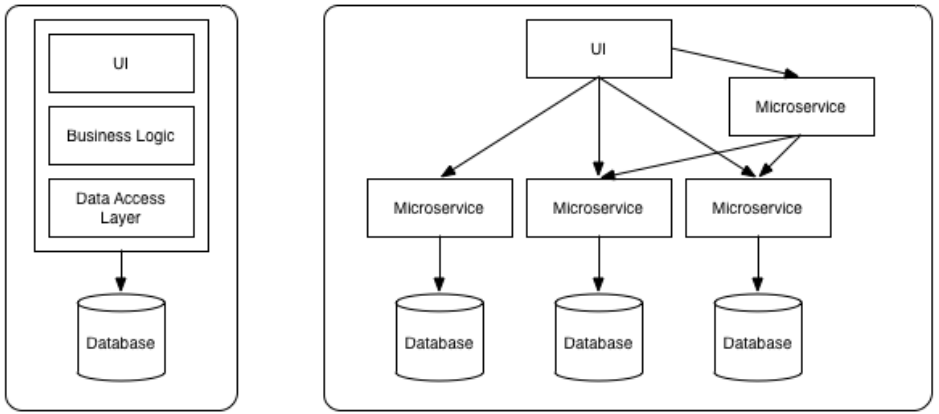
- “A loosely coupled service oriented architecture with bounded contexts” – Adrian Cockcroft
  - Loosely coupled
    - Services can be updated independently
  - Bounded context
    - Services are responsible for a well defined business function and care very little about the other services that surround them
    - ie. “Do one thing and do it well”

Microservices

- Microservice Architectures are
  - Mostly HTTP based
    - May communicate via other open standards too
  - Containerized
  - Independently deployable and scalable
  - Self-sufficient
    - Makes as little assumptions as possible on the external environment

Microservices

- Microservices usually communicate using REST API.
  - So, they are REST Services.
- In Spring Cloud, a microservice is a Spring Boot application with REST controllers.



Monolithic Architecture

Microservices Architecture

Service Discovery

Service Discovery

- A service registry is a phone book for your microservices.
- Each service registers itself with the service registry and tells the registry where it lives (host, port, node name)
  - and perhaps other service-specific metadata - things that other services can use to make informed decisions about it.
- Clients can ask questions about the service topology
  - are there any ‘fulfillment-services’ available, and if so, where?

Service Discovery

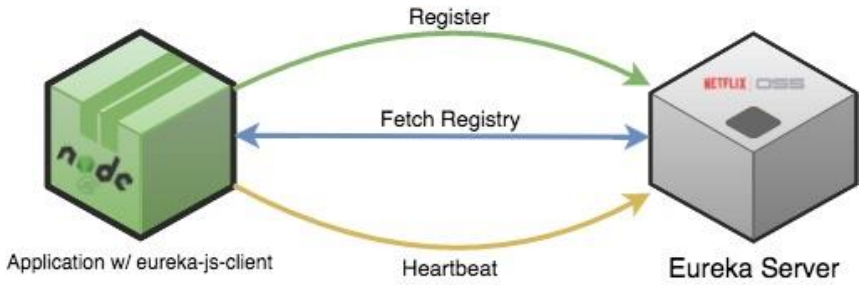
- The discovery server implementation from Netflix is called Eureka.
  - **Eureka Server** is a REST (Representational State Transfer) based service that is primarily used for locating services for the purpose of load balancing and failover of middle-tier servers.
  - Eureka also comes with a Java-based client component, the **Eureka Client**, which makes interactions with the service much easier.

Service Discovery

- The client also has a built-in load balancer that does basic round-robin load balancing.
  - At Netflix, a much more sophisticated load balancer wraps Eureka to provide weighted load balancing based on several factors like traffic, resource usage, error conditions etc. to provide superior resiliency.
  - More importantly, if your architecture fits the model where a client based load balancer is favored, Eureka is well positioned to fit that usage.

Service Discovery

- Eureka has a 30 second heartbeat agreement between servers and clients by default.



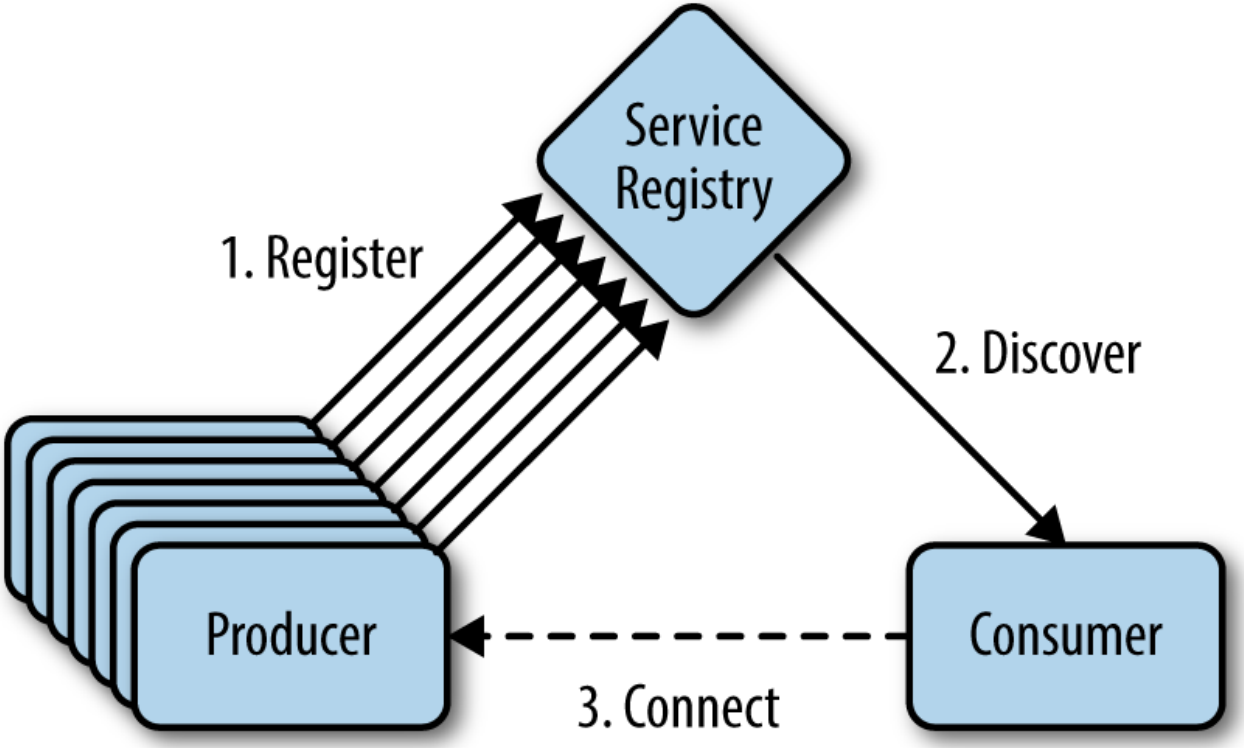
- Clients:
  - register at startup;
  - send a heartbeat every 30 seconds indicating they are alive;

Service Discovery

- The Eureka server does not have a backend store, but this can be done in memory.
  - The service instances in the registry all have to send heartbeats to keep their registrations up to date
- Clients also have an in-memory cache of eureka registrations
  - So they don’t have to go to the registry for every single request to a service.

Service Discovery

- How it works



Service Discovery

- From Spring Cloud point of view Eureka is an embeddable discovery server.
- To create an Eureka deployment you should do three things:
  - Declare project dependencies (pom.xml)
  - Create the main class (SpringBoot configuration class with main method)
  - Create the spring cloud configuration file (main/resources/bootstrap.yml)

Service Discovery – pom.xml (part 1)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.luxoft.training.spring.cloud</groupId>
    <artifactId>eureka</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.2.RELEASE</version>
    </parent>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Camden.SR3</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

Service Discovery – pom.xml (part 2)

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

</project>
```

Service Discovery – Java class

```
package com.luxoft.training.spring.cloud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

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

Service Discovery – bootstrap.yml

```
spring:
  application:
    name: Eureka

server:
  port: 8761

eureka:
  client:
    fetchRegistry: false
    registerWithEureka: false
```


LUXOFT

Service Discovery

- Actually, Spring Cloud applications can work with different discovery servers. For instance, ZooKeeper, Consul and Etcd.
  - If you already use another discovery server which you want to use for Spring Cloud projects too, it is enough to use EnableDiscoveryClient annotation instead of EnableEurekaClient.
  - However, you will still need to add client dependency for desired discovery server to classpath.

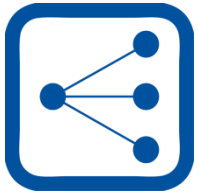
Client-Side Load Balancer

Client-Side Load Balancer

- There are two kinds of load balancers, server-side and client-side.
- The difference is that
  - Server-side load balancing requires additional components to be installed and configured in front of your services.
  - Client-side load balancing requires you to implement the logic at each service client.
- Spring Cloud Netflix introduces integrated client-side load balancing.

Client-Side Load Balancer

- Spring Cloud Netflix provides an integrated client-side load balancer – the **Ribbon**.
- Ribbon is a Inter Process Communication (remote procedure calls) library with built in software load balancers.
  - The primary usage model involves REST calls with various serialization scheme support.

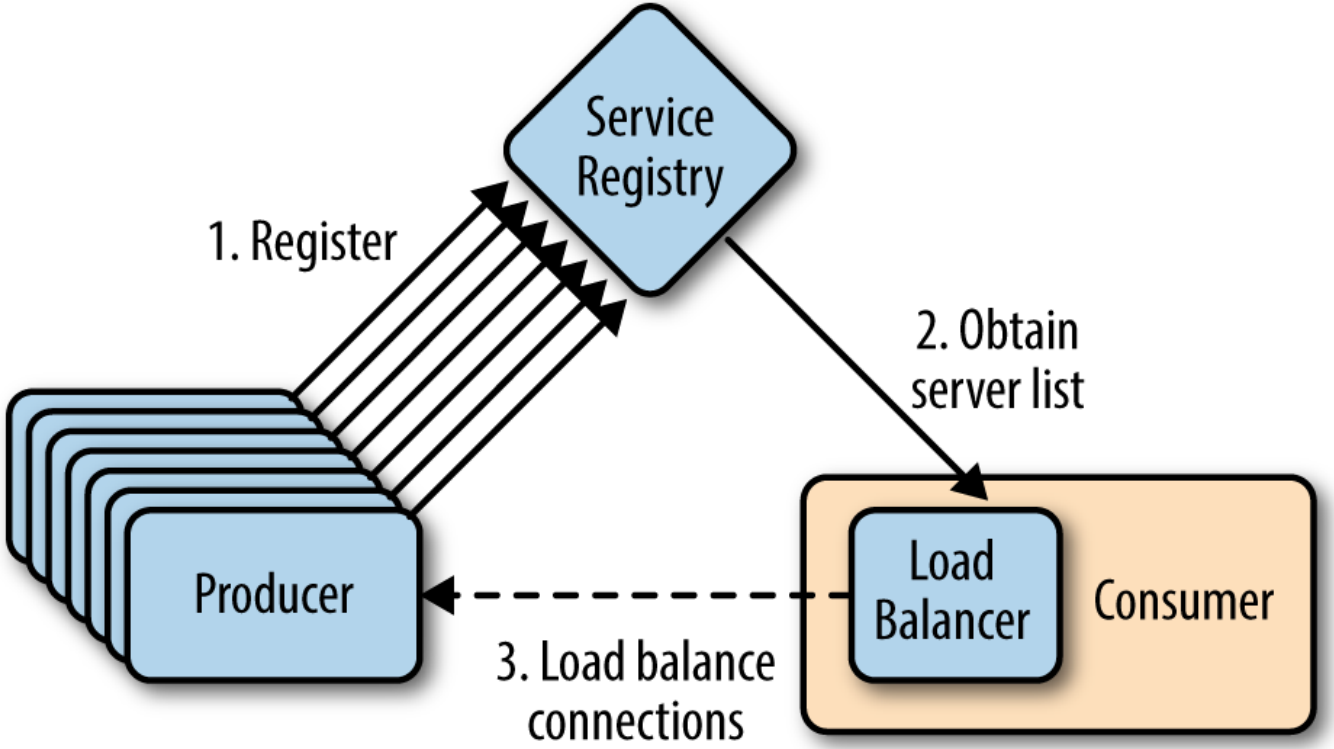


Client-Side Load Balancer

- Whenever a client (microservice) or any consumer talks to Eureka:
  - Eureka will return all instances of service requested;
  - Ribbon will pick next instance according to pre-configured load-balancing logic;
    - By default it uses Round-Robin logic which is the fastest - with lowest latency.

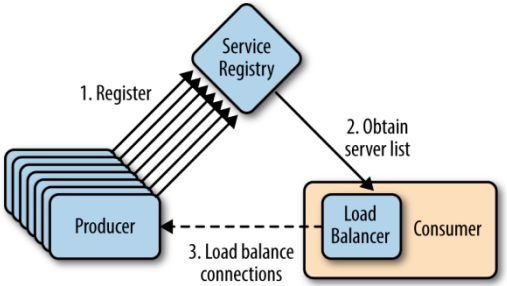
Client-Side Load Balancer

- How it works



Client-Side Load Balancer

- Some of available configuration options are:
  - client-name.ribbon.MaxAutoRetries: 1
  - client-name.ribbon.MaxAutoRetriesNextServer: 1
  - client-name.ribbon.ServerListRefreshInterval: 2000
  - client-name.ribbon.ConnectTimeout: 3000
  - client-name.ribbon.ReadTimeout: 3000
- “client-name” is the name of the service used with REST client.



Client-Side Load Balancer

- A Ribbon client typically load balances between the list of hosts that are fetched from a provider.
  - A provider could be a list of known hosts from configuration file or discovery server.
- We are going to use Ribbon integrated into REST Client, as we are going to make REST calls to registered microservices.
  - Load balancing using fixed list of hosts is out of scope of this training.

Client-Side Load Balancer

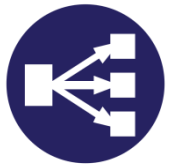
- To use Ribbon with RestTemplate add dependency

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

- Declare RestTemlate with @LoadBalanced annotation

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

- To use it with discovery server, just use service-name instead of hostname.
  - For instance, **http://ServiceName/create/**



REST Client

REST Client

- Developing a REST client is not difficult.
  - However, for REST calls between microservices you should integrate with discovery server and load balance between available instances.
- Spring Cloud Netflix provides **Feign** – a declarative REST client integrated with Eureka and Ribbon
  - Feign makes writing java http clients easier.

REST Client

- Feign aims to connect your code to http APIs with minimal overhead and code.
  - Spring Cloud follows declarative approach to implement Feign REST clients.
- Feign works by processing annotations into a templatized request.
  - Just before sending it off, arguments are applied to these templates in a straightforward fashion.
  - While this limits Feign to only supporting text-based APIs, it dramatically simplified system aspects such as replaying requests.

REST Client

- Before creating a Feign client you should
  - Add dependency

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

  - Add **`@EnableFeignClients`** annotation to your configuration class

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class SomeService {
    //main() method
}
```

REST Client

- To create a Feign client you need to create the interface only, implementation will be injected by container.

```
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import java.util.List;

@FeignClient("Service")
public interface ServiceClient {
    @RequestMapping("/get/{parent_id}")
    List<Item> getItems(@PathVariable("parent_id") Integer parentId);
}
```

REST Client

- So, when you actually create a Feign instance,
  - it will automatically lookup for Eureka discovery client configuration and load balance between instances of microservices using Ribbon client-side load balancer.
- Ribbon handles all remote-call related tasks
  - Including communication with discovery servers to fetch the instances and select which one of them will be used by Feign according to load-balancing logic.

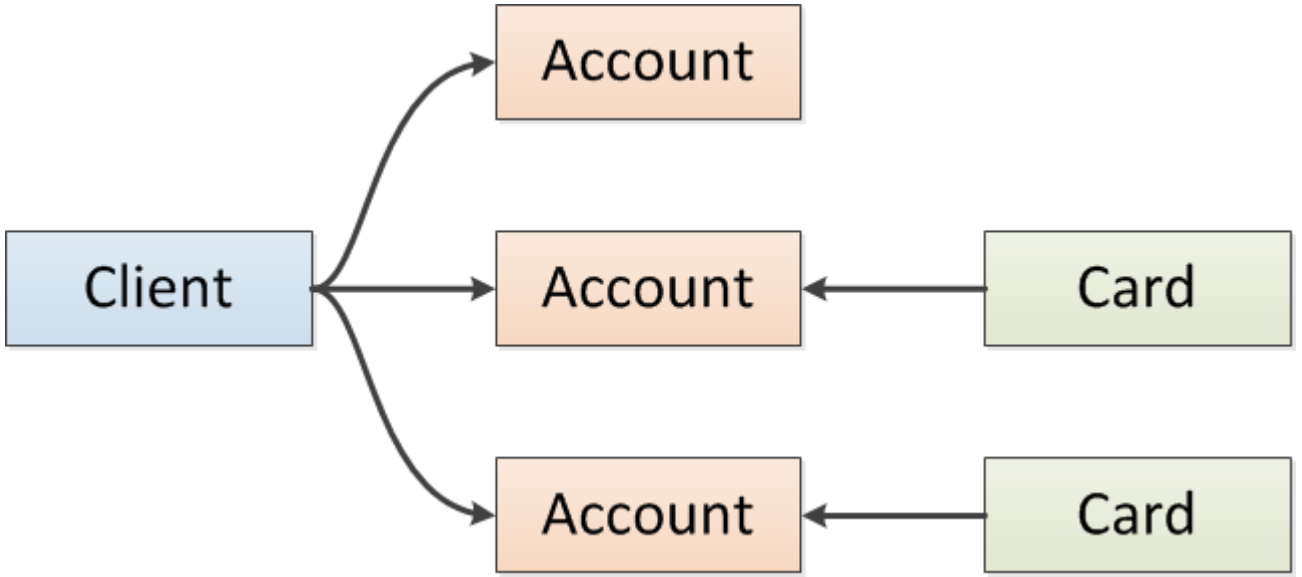
Lab 1

Banking Application

- Story:
  - We are going to develop banking software with microservices approach. Clients open accounts in bank, order plastic cards and do checkout and fund operations using different terminals. Internet banking and other internal projects should also reuse existing infrastructure.
- We have several entities, namely
  - Client
  - Account
  - Card

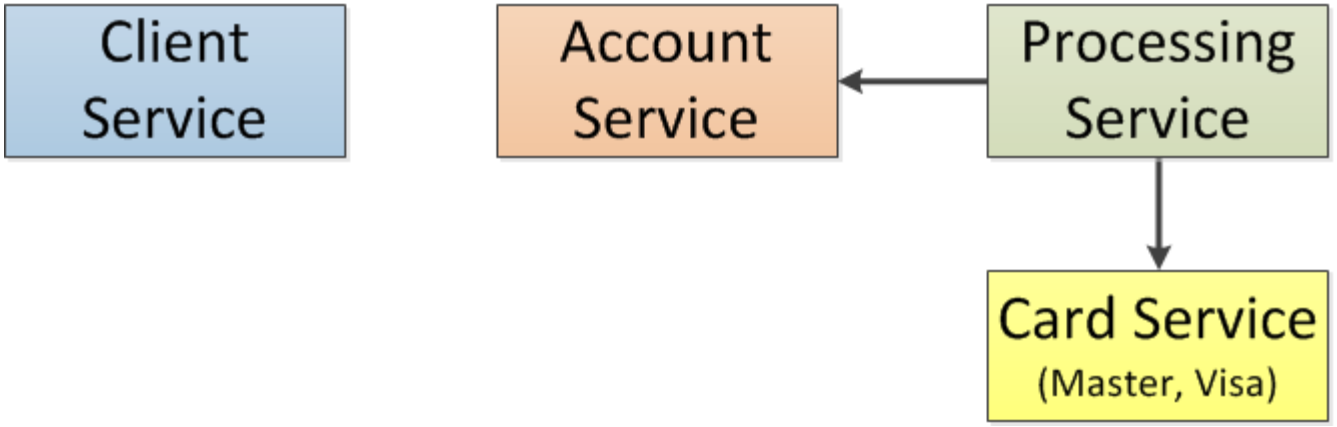
Banking Application

- Entity relationship



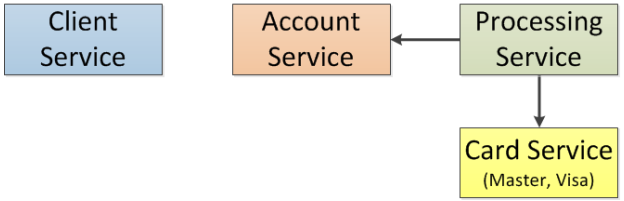
Banking Application

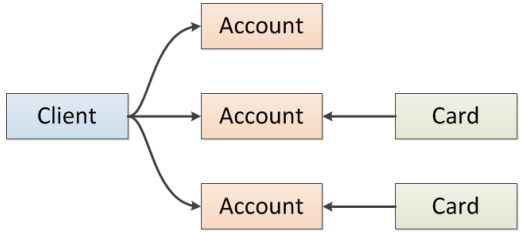
- Service topology



Banking Application

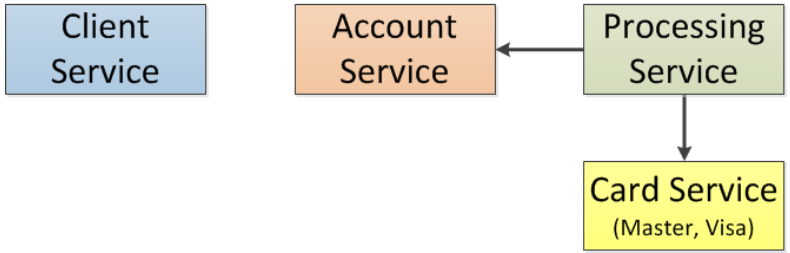
- Our microservices will be:
  - Client Service – operations with bank clients
  - Account Service – operations with client accounts
  - Card Service – external service (MasterCard, Visa)
  - Processing Service – operations with card accounts





Banking Application

- Each of these services is a separate physical service and can reuse another services available to achieve it’s goals.
- Services should expose REST API and talk each other using REST.



Banking Application

- We have to develop 5 applications, namely
  - Discovery Server
  - Client Service, Account Service, Card Service, Processing Service
- Netflix components to be used are:
  - Eureka
  - Ribbon
  - Feign

Banking Application

- A few conventions to be able to test REST using web browser
  - Use GET methods
  - Use path and request parameters

Bottlenecks & Issues

Discovery Server

- In such architectures discovery server can be a bottleneck.
  - If it gets unreachable, all services depending on it will fail.
- Eureka should always be available, if it goes down, services using it to lookup for other services will fail.
  - Having one instance of Eureka is a bottleneck.

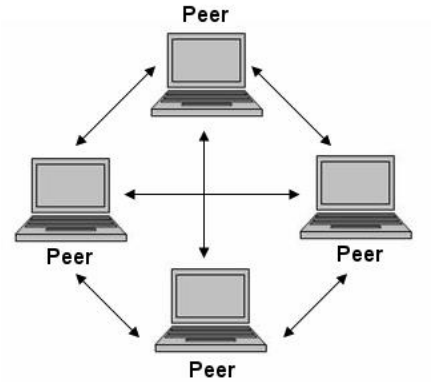
Discovery Server Bottleneck

Discovery Server

- Eureka benefits from the resiliency built into both the client and the servers.
- By default every Eureka server is also an Eureka client and requires at least one service URL to locate a peer.
  - If you don’t provide it, the service will run and work trying to lookup for default peer, and if won’t find it, it will shower your logs with a lot of noise about not being able to register with the peer.

Discovery Server

- Two or more instances of Eureka can be pointed to each other for peer to peer communications.
  - This will sync all services registered at available instances.



Discovery Server

```
spring:
  application:
    name: Eureka

server:
  port: 8701

eureka:
client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://host2:8702/eureka
```

```
spring:
  application:
    name: Eureka

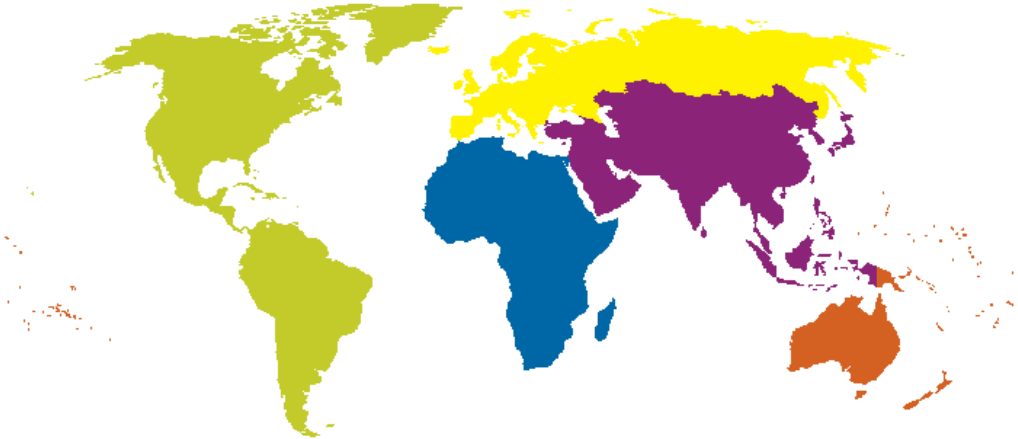
server:
  port: 8702

eureka:
client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://host1:8701/eureka
```

In such cases when more than two instances should be mirrored, list them in a comma separated fashion.

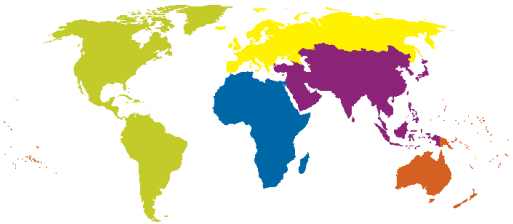
Discovery Server

- For more availability, the whole surface of Eureka servers can be split to regions, furthermore, regions can be split to availability zones.
  - Eureka clusters between regions do not communicate with one another.



Discovery Server

- The first information you would configure in Eureka Servers is the availability zones for a region.
- In the following example, a region *us-east-1* is specified to have 3 availability zone *us-east-1c,us-east-1d, us-east-1e*.
  - eureka.us-east-1.availabilityZones: us-east-1c,us-east-1d,us-east-1e



Discovery Server

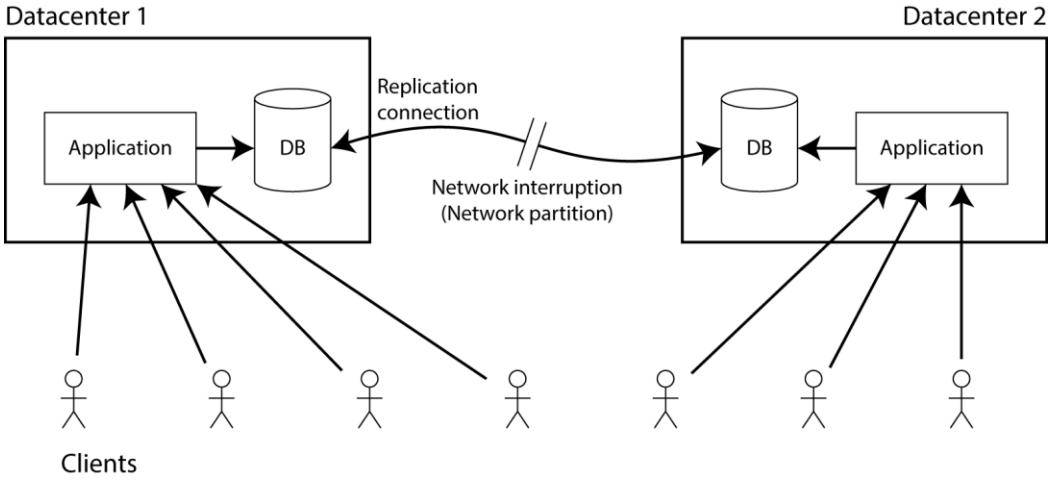
- Next, you configure is the service URLs for each zone where Eureka is listening for requests. Multiple eureka servers for a zone can be configured by providing a comma-delimited list.
  - eureka.serviceUrl.us-east-1c: http://host165:7001/eureka/v2/,http://host134:7001/eureka/v2/
  - eureka.serviceUrl.us-east-1d: http://host170:7001/eureka/v2/
  - eureka.serviceUrl.us-east-1e: http://host592:7001/eureka/v2/
- The same configuration is then included with Eureka clients that register with Eureka Service and also with Eureka client that want to find the services from Eureka Server.

Discovery Server

- Eureka clients are built to handle the failure of one or more Eureka servers.
  - Since Eureka clients have the registry cache information in them, they can operate reasonably well, even when all of the eureka servers go down.

Discovery Server

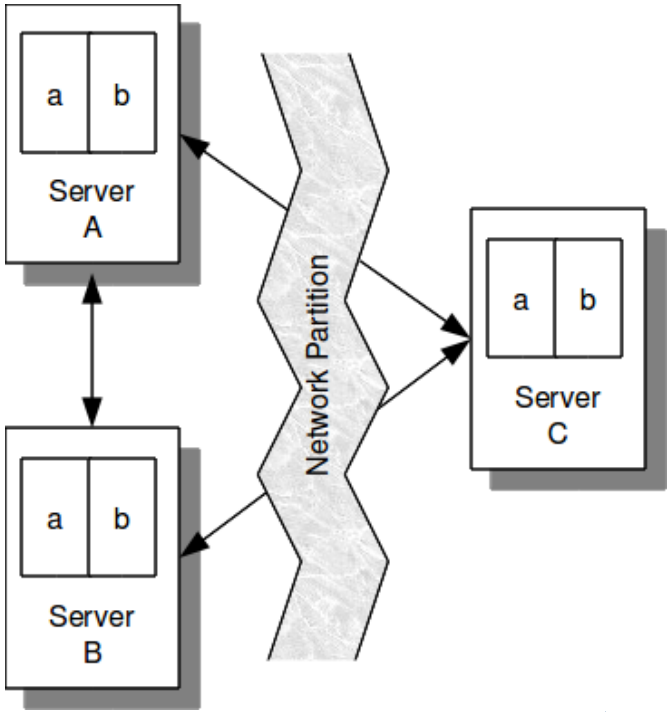
- Eureka Servers are resilient to other eureka peers going down.
  - Even during a network partition between the clients and servers, the servers have built-in resiliency to prevent a large scale outage.



Network Partition Issue

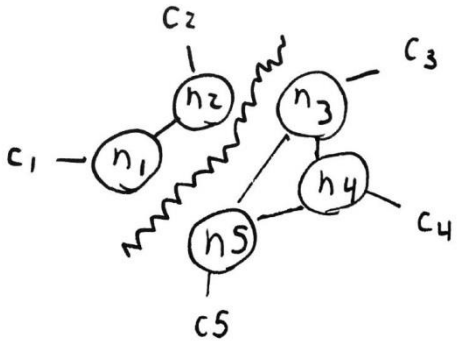
Network Partition

- A network partition refers to the failure of a network device that causes a network to be split.
- For instance,
  - Imagine a network with servers A, B and C
  - Network can be split between servers
  - A and B still see each other
  - However, C is not reachable



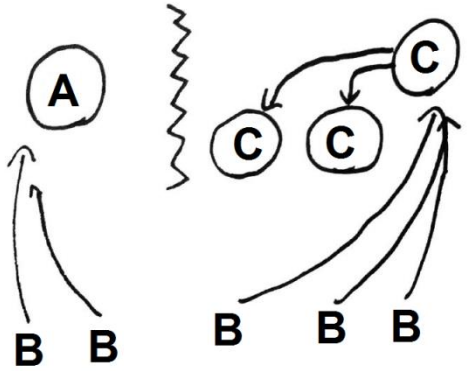
Network Partition

- In such cases, it can happen so that Eureka will consider some services down as they are unreachable. What to do then?
  - Forget unreachable services?
  - May be, Eureka cannot reach some services, but on the other side of network they are still available for other consumers.
- Eureka has so called self-preservation mode.
  - Eureka does not forget any services registered before
  - Even if they are not reachable or down at all



Network Partition

- Self-preservation mode addresses network partition issue
  - It’s better to have something than nothing at all
  - B can still be able to reach A and C both, when A and C are partitioned.
  - Here A could be Eureka, C and B could be some service.



Network Partition

- By default this mode is turned on
  - If you want unreachable services to be unregistered, self-preservation mode could be turned off
- To control self-preservation mode use bootstrap configuration
  - ```
    eureka:
      server:
        enable-self-preservation: false
    ```
    ```

Network Partition

Examples from previous slide can be modified like:

```
spring:
  application:
    name: Eureka

server:
  port: 8701

eureka:
  server:
    enable-self-preservation: false
  client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://host2:8702/eureka
```

```
spring:
  application:
    name: Eureka

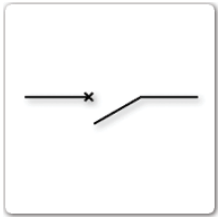
server:
  port: 8702

eureka:
  server:
    enable-self-preservation: false
  client:
    fetchRegistry: true
    registerWithEureka: true
    serviceUrl:
      defaultZone: http://host1:8701/eureka
```

Circuit Breaker

Circuit Breaker

- A circuit breaker is a simple structure that monitor faults.
- It helps to achieve resilience.
  - Resilience is the ability of the network to provide and maintain an acceptable level of service in the face of various faults and challenges to normal operation.



Circuit Breaker

- The circuit breaker identifies long waiting times among the calls to the vendor and fails-fast, returning an error response instead of making the threads wait.
  - Thus, the circuit breaker helps to achieve better response times.

Circuit Breaker

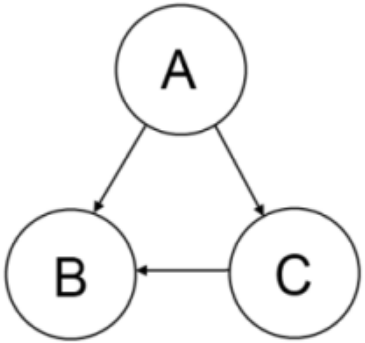
- In a distributed environment, inevitably some of the many service dependencies will fail.
- Hystrix is a library that helps you control the interactions between these distributed services by adding latency tolerance and fault tolerance logic.
  - Hystrix does this by isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve your system's overall resiliency.

Circuit Breaker

- Hystrix is designed to do the following:
  - Give protection from and control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.
  - Stop cascading failures in a complex distributed system.
  - Fail fast and rapidly recover.
  - Fallback and gracefully degrade when possible.
  - Enable near real-time monitoring, alerting, and operational control.

Circuit Breaker

- Applications in complex distributed architectures have dozens of dependencies, each of which will inevitably fail at some point.
  - If the host application is not isolated from these external failures, it risks being taken down with them.



Circuit Breaker

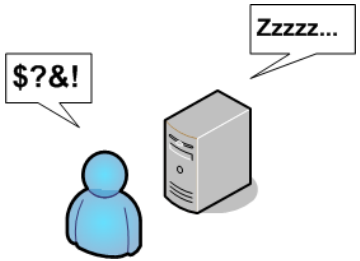
- For example, for an application that depends on 30 services where each service has 99.99% uptime, here is what you can expect:
  - 99.99^{30} = 99.7% uptime

    0.3% of 1 billion requests = 3,000,000 failures

    2+ hours downtime/month even if all dependencies have excellent uptime.
    - Reality is generally worse!

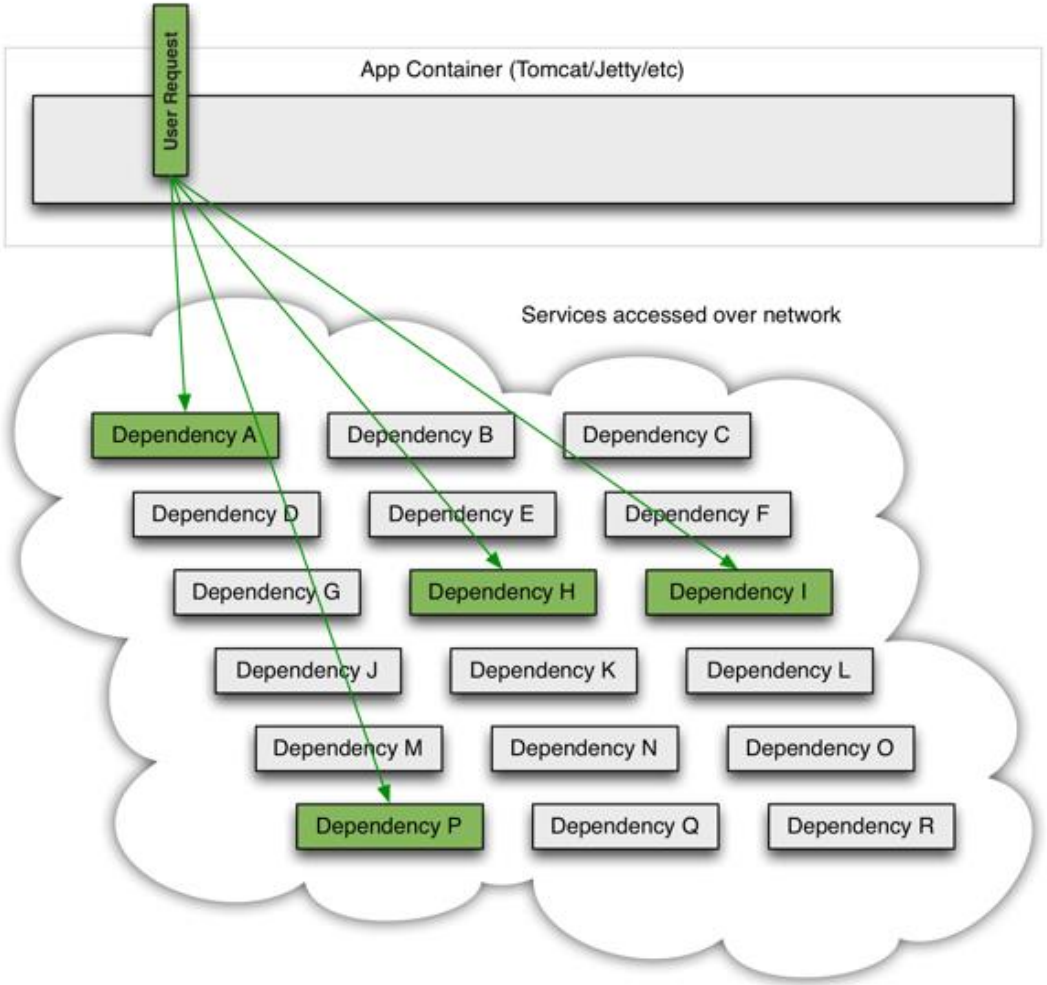
Circuit Breaker

- Even when all dependencies perform well the aggregate impact of even 0.01% downtime on each of dozens of services equates to potentially hours a month of downtime if you do not engineer the whole system for resilience.
  - With high volume traffic a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers.



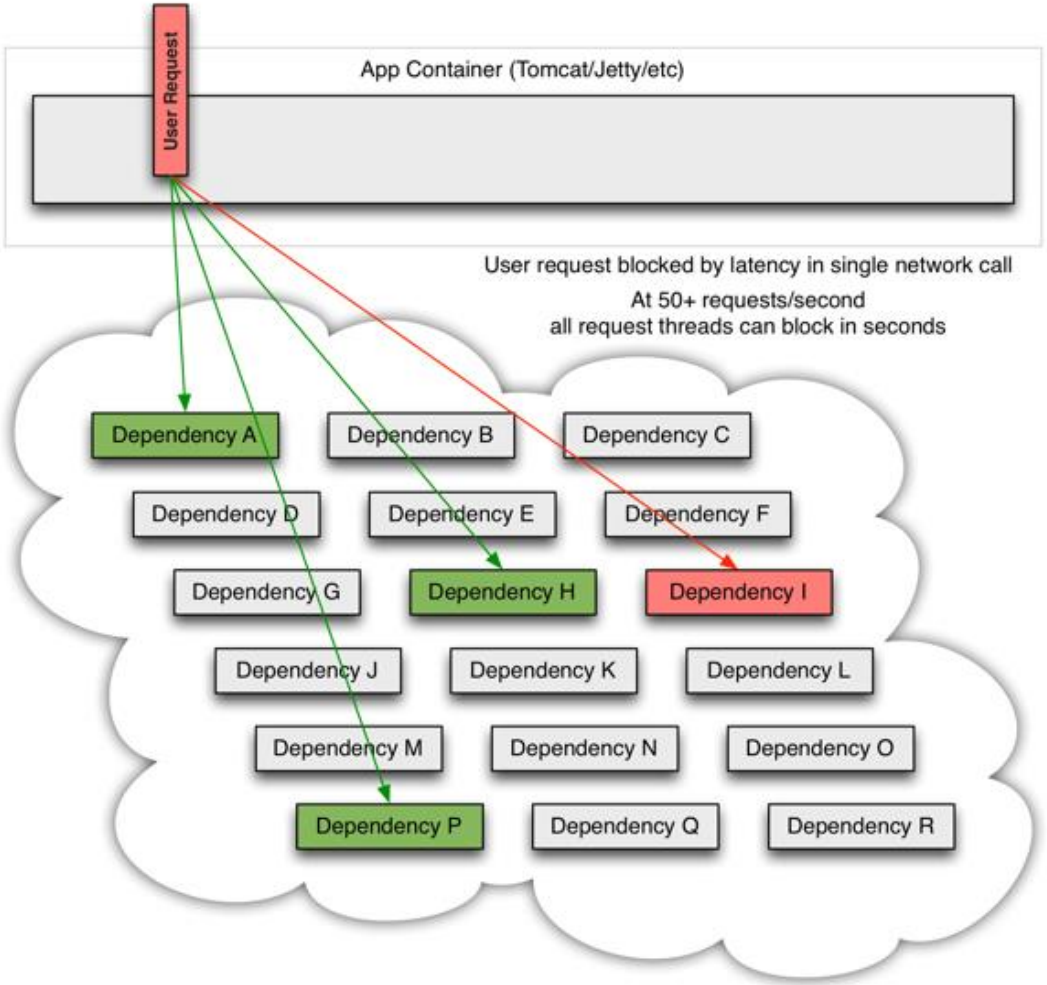
Circuit Breaker

- When everything is healthy the request flow can look like this:



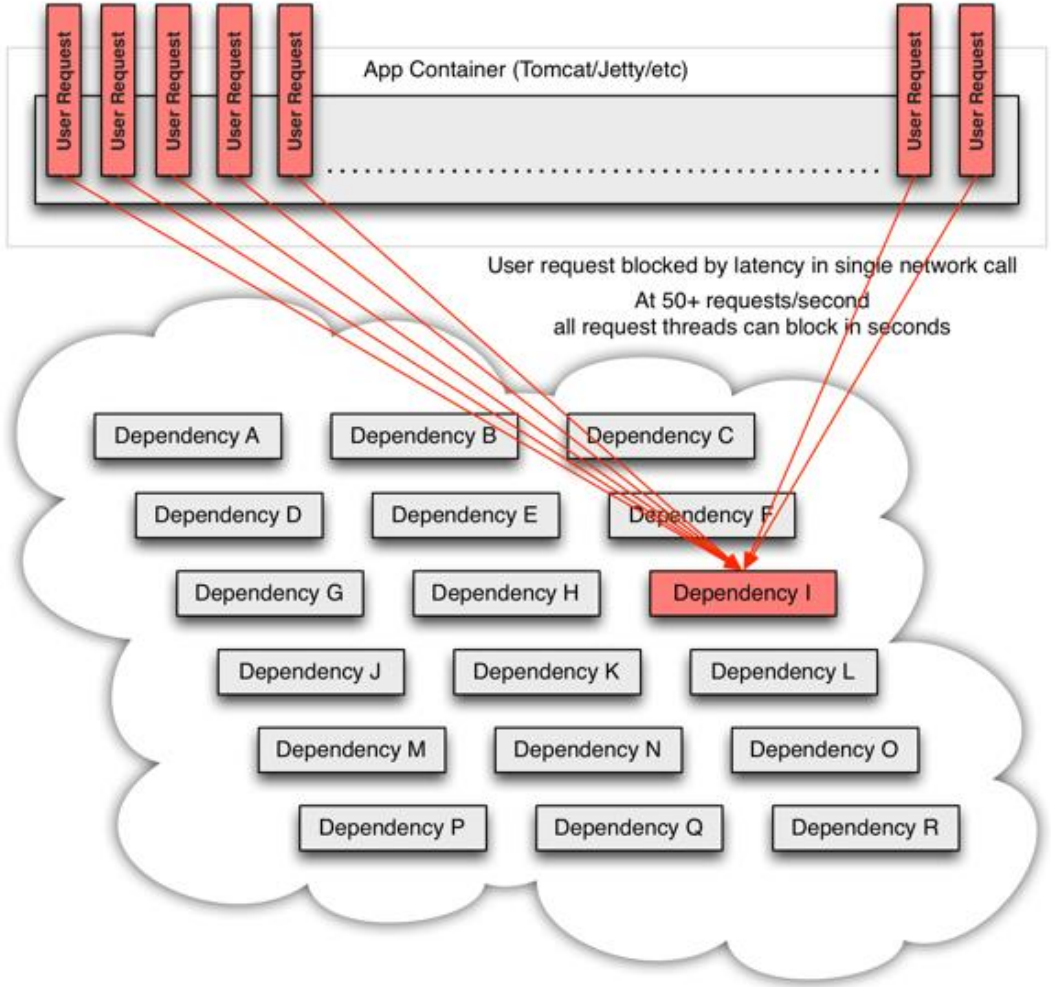
Circuit Breaker

- When one of many backend systems becomes latent it can block the entire user request.



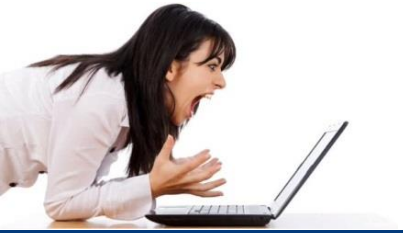
Circuit Breaker

- Every point in an application that reaches out over the network or into a client library that might result in network requests is a source of potential failure.
- Worse than failures, these applications can also result in increased latencies between services, which backs up queues, threads, and other system resources causing even more cascading failures across the system.



Circuit Breaker

- These issues are exacerbated when network access is performed through a third-party client — a “black box” where implementation details are hidden and can change at any time, and network or resource configurations are different for each client library and often difficult to monitor and change.
  - Even worse are transitive dependencies that perform potentially expensive or fault-prone network calls without being explicitly invoked by the application.



Circuit Breaker

- Network connections fail or degrade, services and servers fail or become slow, new libraries or service deployments change behavior or performance characteristics, client libraries have bugs.
- All of these represent failure and latency that needs to be isolated and managed so that a single failing dependency can’t take down an entire application or system.

Circuit Breaker

- Hystrix works by:
  - Preventing any single dependency from using up all container (such as Tomcat) user threads.
  - Shedding load and failing fast instead of queueing.
  - Providing fallbacks wherever feasible to protect users from failure.
  - Using isolation techniques (such as bulkhead, swimlane, and circuit breaker patterns) to limit the impact of any one dependency.

Circuit Breaker

- Hystrix works by (continuation):
  - Optimizing for time-to-discovery through near real-time metrics, monitoring, and alerting
  - Optimizing for time-to-recovery by means of low latency propagation of configuration changes and support for dynamic property changes in most aspects of Hystrix, which allows you to make real-time operational modifications with low latency feedback loops.
  - Protecting against failures in the entire dependency client execution, not just in the network traffic.

Circuit Breaker

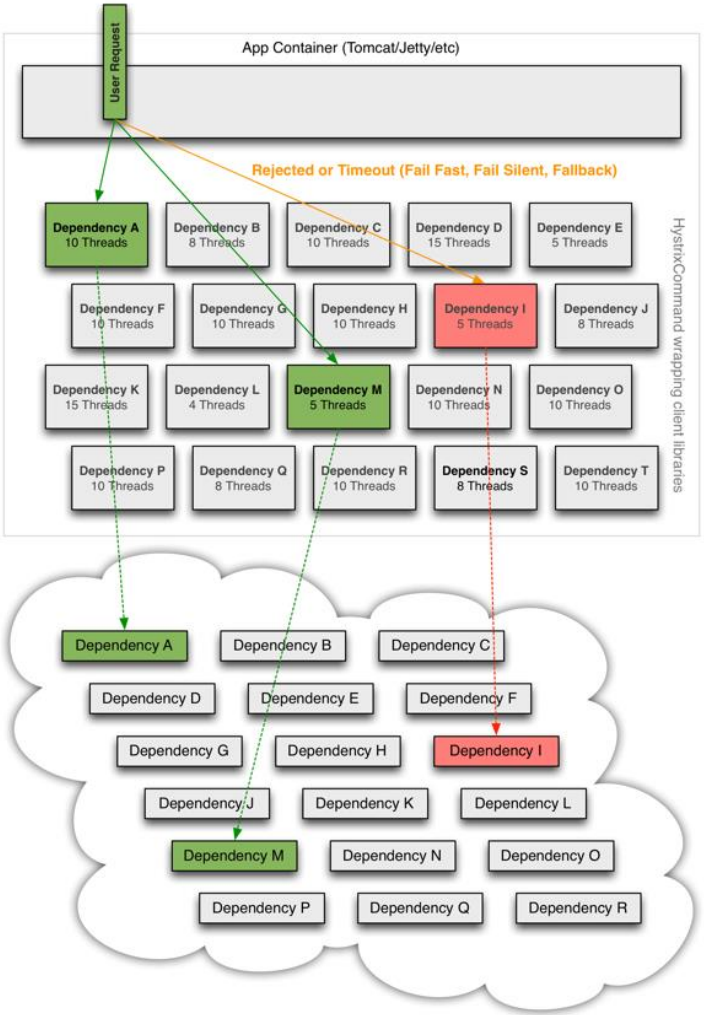
- Hystrix does this by:
  - Wrapping all calls to external systems (or “dependencies”) in an object which typically executes within a separate thread.
  - Timing-out calls that take longer than thresholds you define.
  - Maintaining a small thread-pool (or semaphore) for each dependency; if it becomes full, requests destined for that dependency will be immediately rejected instead of queued up.

Circuit Breaker

- Hystrix does this by (continuation):
  - Measuring successes, failures (exceptions thrown by client), timeouts, and thread rejections.
  - Tripping a circuit-breaker to stop all requests to a particular service for a period of time, either manually or automatically if the error percentage for the service passes a threshold.
  - Performing fallback logic when a request fails, is rejected, times-out, or short-circuits.
  - Monitoring metrics and configuration changes in near real-time.

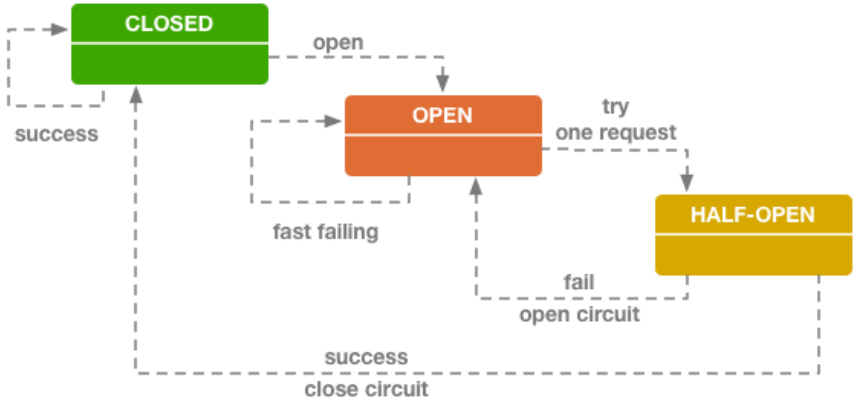
Circuit Breaker

- When you use Hystrix to wrap each underlying dependency, the overall architecture changes.
- Each dependency is isolated from one other, restricted in the resources it can saturate when latency occurs, and covered in fallback logic that decides what response to make when any type of failure occurs in the dependency:



Circuit Breaker

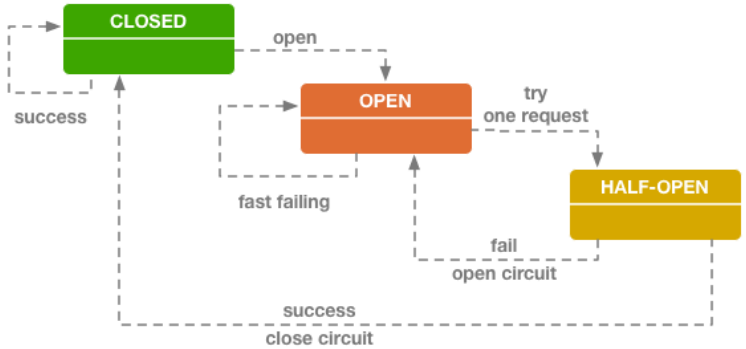
- Hystrix circuit breaker allows to control latency, fault tolerance, isolate points of access to remote systems, services and 3rd party libraries, stop cascading failures, and enable resilience in complex distributed systems where failure is inevitable.



Circuit Breaker State Diagram

Circuit Breaker

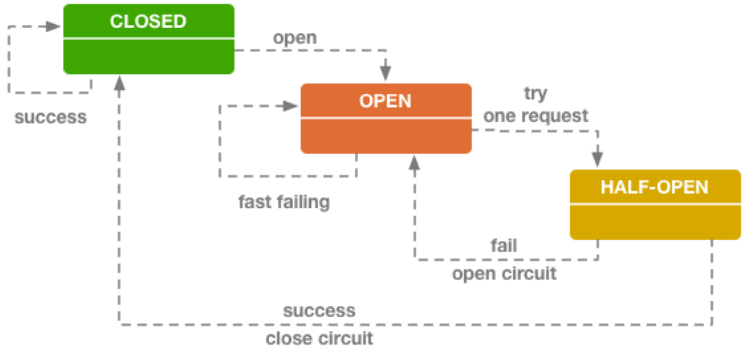
- Normal function (Closed)
  - When a system is functioning smoothly, the resiliency is measured by the state of its success counters, while any failures are tracked using the failure gauges. This design ensures that when the threshold for failures is reached, the circuit breaker opens the circuit to prevent further calls to the dependent resource.



Circuit Breaker State Diagram

Circuit Breaker

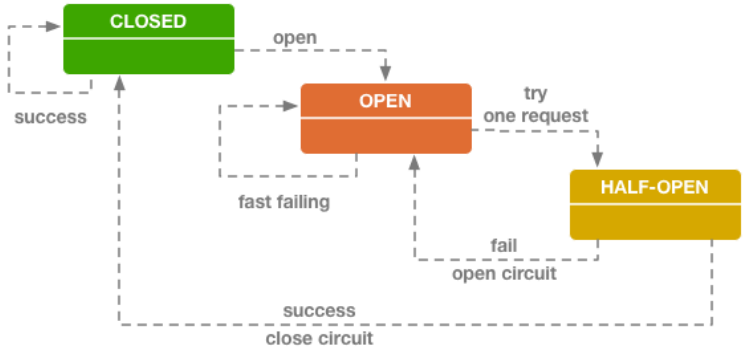
- Failure state (Open)
  - At this juncture, every call to the dependency is short-circuited with a HystrixRuntimeException and FailureType of SHORTCIRCUIT, giving clear indication of its cause. Once the sleepInterval passes, the Hystrix circuit breaker moves into a half-open state.



Circuit Breaker State Diagram

Circuit Breaker

- Half-open state
  - In this state, Hystrix takes care of sending the first request to check system availability, letting other requests fail-fast until the response is obtained. If the call is successful, the circuit breaker is reset to Closed; in case of failure, the system goes back to the Open state, and the cycle continues.



Circuit Breaker State Diagram

Circuit Breaker

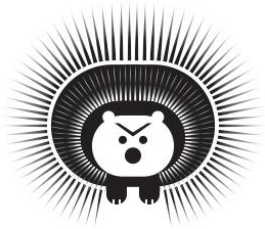
- Feign is already integrated with Hystrix.
- To use Hystrix, implement the client interface and set it as fallback class for Feign.

```
@FeignClient(name = "Service", fallback = ServiceFallback.class)
public interface ServiceClient {
    @RequestMapping("/get/{parent_id}")
    List<Item> getItems(@PathVariable("parent_id") Integer parentId);
}
```

  - ServiceFallback class implements ServiceClient interface

Circuit Breaker

- To configure timeouts, you can use
  - hystrix.command.**default**.execution.isolation.thread.timeoutInMilliseconds: 1000
- This timeout applies to all usages of circuit breaker when used with “default” key.
- For separation of timeouts instead of “default” key you can use Feign client interface with method signature
  - `DemoServiceClient#demoAction()`
  - `AccountServiceClient#checkout(Integer,BigDecimal)`



Lab 2

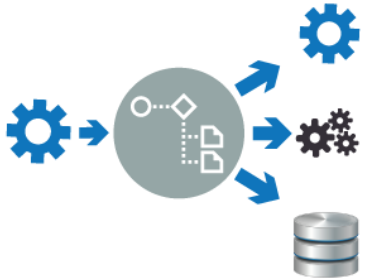
Lab 2 – Bottlenecks & Issues

- In this lab we will
  - adopt Eureka peer to peer communications to create a mirrored-style fault tolerant cluster of discovery servers;
  - see Eureka self-preservation mode in action that addresses network partitions
  - attach fallbacks to Feign clients that trigger by Hystrix circuit breaker

API Gateway

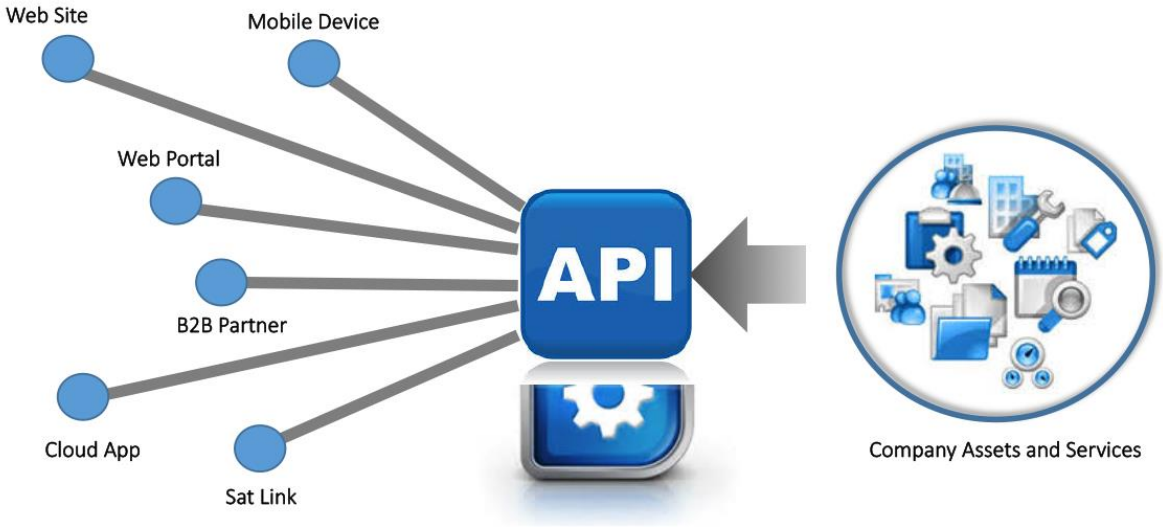
API Gateway

- There are cases, when a client is not aware or should not be aware of your underlying microservices. It may be user, other application, UI and etc.
- These applications should have a single entry point, a gateway that allows them to reuse your microservices.



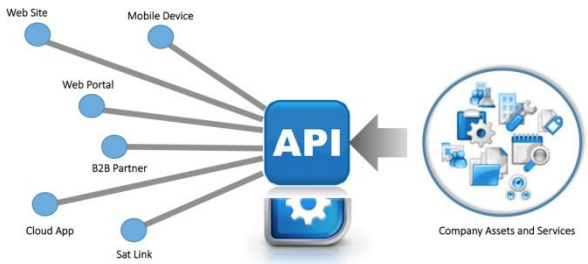
API Gateway

- Having a single entry point, security, routing, caching, orchestration, auditing and other stuff can be implemented here without any latency overhead to the rest of microsystems



API Gateway

- Zuul is the edge proxy from Netflix that allows to implement API gateways.
  - Zuul is the front door for all requests from devices and web sites to the backend of the application.
- As an edge service application, Zuul is built to enable dynamic routing, monitoring, resiliency and security.

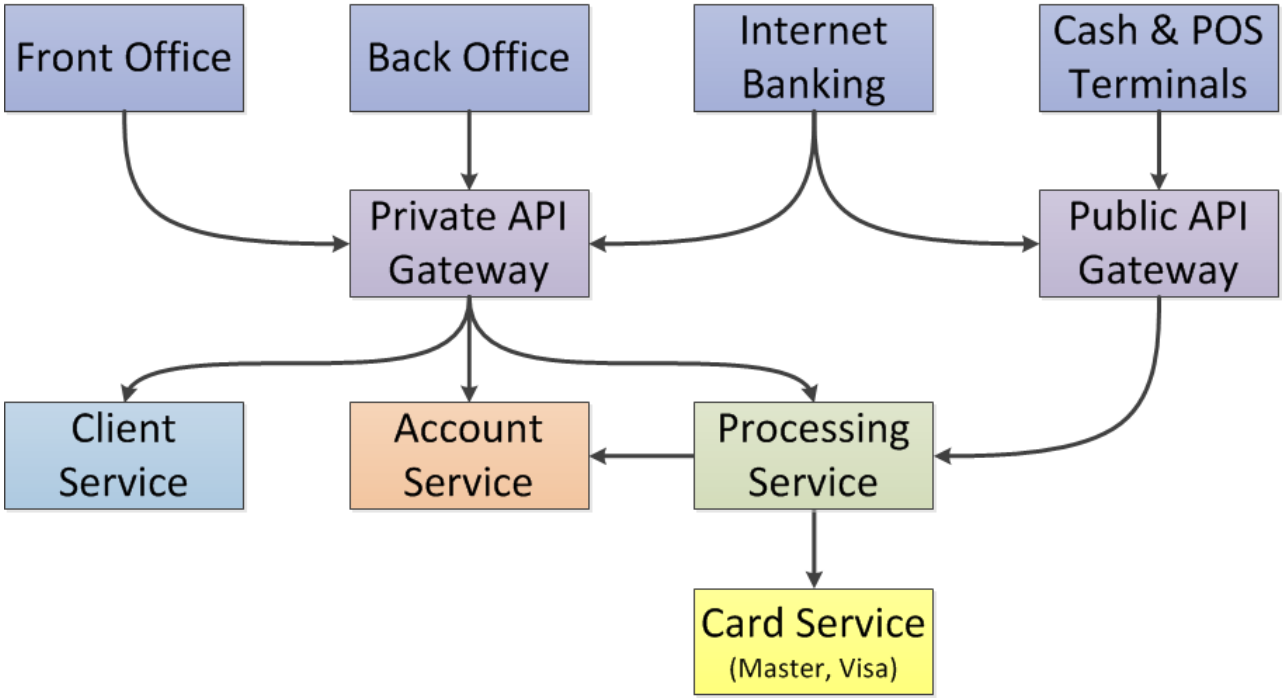


API Gateway

- In Spring Cloud, Zuul is embeddable as Eureka.
- To create a Zuul based API gateway we need to
  - Add dependency spring-cloud-starter-zuul
  - Add annotation @EnableZuulProxy
  - Add Zuul configuration to bootstrap.yml

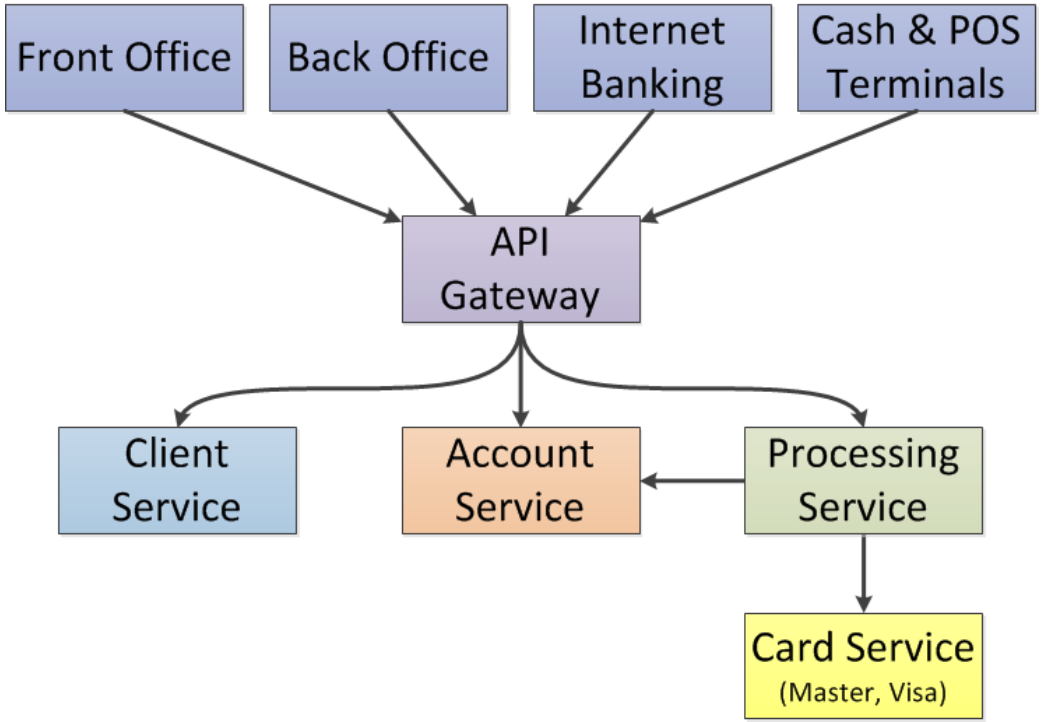
API Gateway

- Using API gateways we can control how clients access our services.



API Gateway

- Using API gateways we can control how clients access our services.



Zuul Configuration

- Available routes can be fetched using **`/routes`** endpoint of Spring Cloud Zuul application.
- Some configuration keys available are:
  - ignoredServices – Used to filter services that should be ignored. Can be used to turn off automatic service discovery which is turned on if you have Eureka client available (in classpath)
  - path – Path for the route
  - serviceId – Service name registered at discovery server
  - stripPrefix – Remove or leave the route “path” when the request will be forwarded to original service

Zuul Configuration

- An example bootstrap.yml configuration part for Zuul may look like:

```
zuul:
  ignoredServices: '*'
  routes:
    clients:
      path: /client/**
      serviceId: ClientService
      stripPrefix: true
    accounts:
      path: /account/**
      serviceId: AccountService
      stripPrefix: true
    card-processing:
      path: /processing/**
      serviceId: ProcessingService
      stripPrefix: true
```

Zuul Configuration

- Additionally, any URL can be mapped as a route using

```
get:
  path: /get/**
  url: http://httpbin.org/get
```

- Using this approach, you can point Zuul to any web application in the network, not necessarily a service registered at discovery server.

Zuul Configuration

- Among other settings available there are:
  - **`sensitiveHeaders: Cookie,Set-Cookie,Authorization`**
    - Should be provided for each route separately.
    - The sensitiveHeaders are a blacklist and the default is not empty, so to make Zuul send all headers (except the "ignored" ones) you would have to explicitly set it to the empty list. This is necessary if you want to pass cookie or authorization headers to your back end.

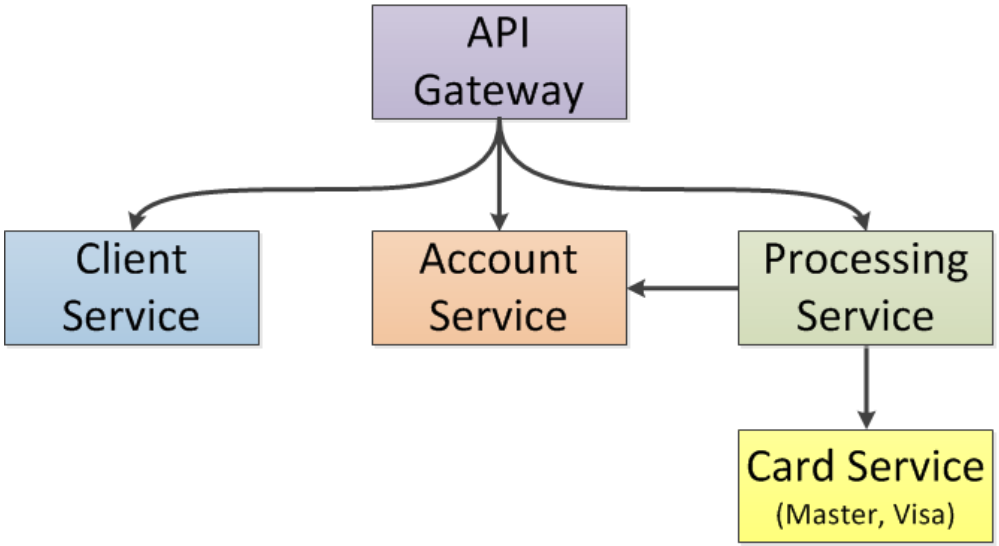
Zuul Configuration

- Among other settings available there are (continuation):
  - In addition to the per-route sensitive headers, you can set a global value for **`zuul.ignoredHeaders`** for values that should be discarded (both request and response) during interactions with downstream services.
  - By default these are empty, if Spring Security is not on the classpath, and otherwise they are initialized to a set of well-known "security" headers (e.g. involving caching) as specified by Spring Security.
    - The assumption in this case is that the downstream services might add these headers too, and we want the values from the proxy. To not discard these well known security headers in case Spring Security is on the classpath you can set **`zuul.ignoreSecurityHeaders`** to **`false`**. This can be useful if you disabled the HTTP Security response headers in Spring Security and want the values provided by downstream services

Lab 3

Lab 3 – API Gateway

- In this lab we will create an API gateway using Zuul edge proxy



Centralized Configuration

Centralized Configuration

- Configuration of microservices can change from time to time.
- To edit configuration files in a more comfortable way from a centralized repository would be a good option.
  - For this purpose we can name all bootstrap files by name of application and put into a single shared storage that could be available to all our microservices.
- But file storage lacks versioning, review and audit abilities.

Centralized Configuration

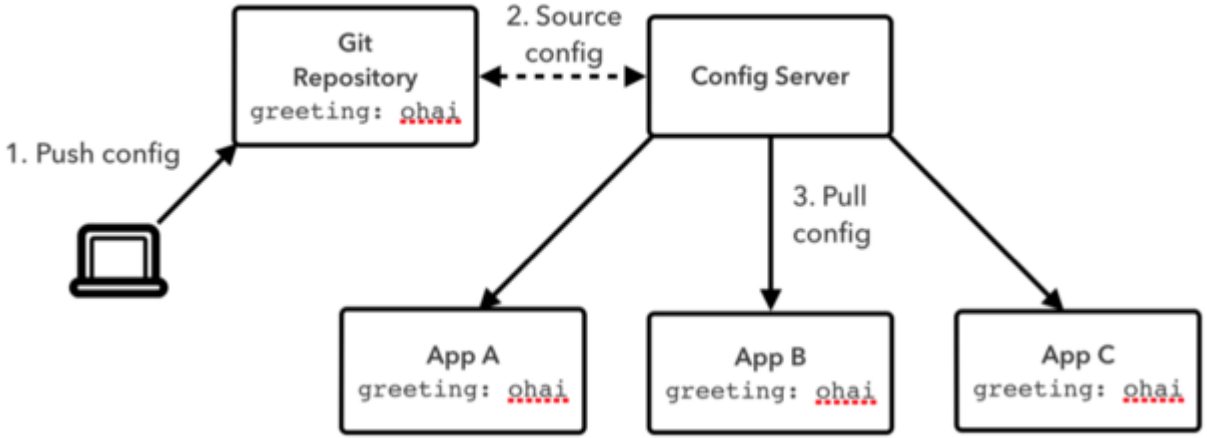
- Spring Cloud Config project provides server and client-side support for externalized configuration in a distributed system.
  - With the Config Server you have a central place to manage external properties for applications across all environments.
- The concepts on both client and server map identically to the Spring Environment and PropertySource abstractions, so they fit very well with Spring applications.

Centralized Configuration

- As an application moves through the deployment pipeline from dev to test and into production you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate.
  - The default implementation of the server storage backend uses Git so it easily supports labeled versions of configuration environments, as well as being accessible to a wide range of tooling for managing the content.

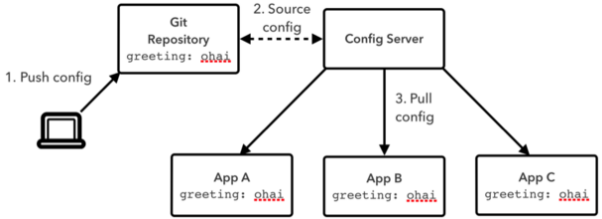
Centralized Configuration

- Spring Cloud Config project allow to turn a Git, SVN or a File repository to a centralized configuration storage.
- Spring Cloud Config is integrated to Spring Cloud infrastructure.



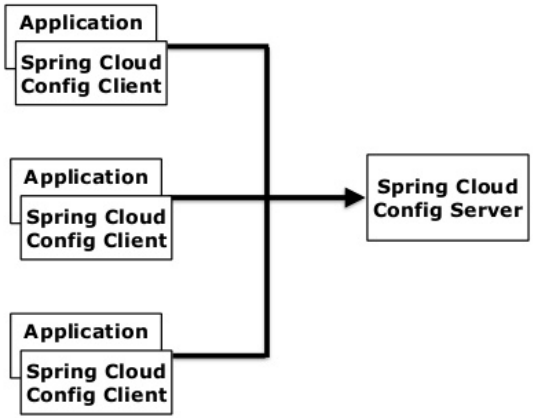
Centralized Configuration

- All standard Spring Boot externalized configuration options are supported
  - a property can be marked with **`@Value("${config.name}"`**) annotation to link the property to a configuration value on initialize;
  - If Spring Boot Actuator is active, then a bean with **`@Value`** properties can be marked with **`@RefreshScope`** annotation to refresh values without restarting the application server whenever the /refresh endpoint of actuator is triggered;



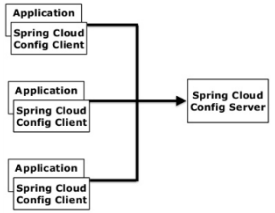
Centralized Configuration

- To configure configuration-client you need to
  - Add **`spring-cloud-starter-config`** dependency
  - Configure property **`spring.cloud.config.uri`**
    - default value is
      - `http://localhost:8888`



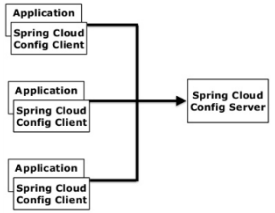
Centralized Configuration

- In case if you are not sure that configuration server will be available at service startup (e.g. all services start together), you can use
  - **`spring.cloud.config.failFast: true`**
    - Fail if configuration server is not available (default false)
  - **`spring.cloud.config.retry.initialInterval: 1000`**
    - Initial interval in milliseconds to wait before retry
  - **`spring.cloud.config.retry.maxAttempts: 6`**
    - Max retry attempts



Centralized Configuration

- In case if you are not sure that configuration server will be available at service startup (e.g. all services start together), you can use (continuation…)
  - **`spring.cloud.config.retry.maxInterval: 2000`**
    - Interval in milliseconds between each retry
  - **`spring.cloud.config.retry.multiplier: 1.1`**
    - Multiplier of interval with each retry

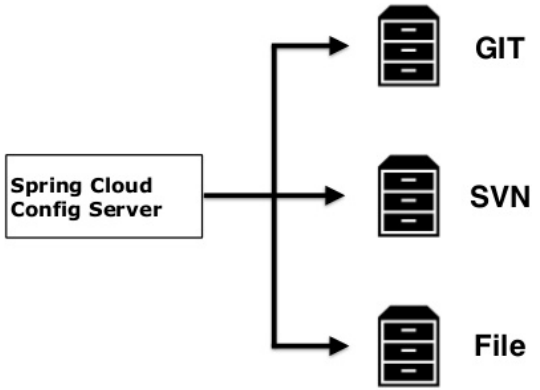


Centralized Configuration

- If you prefer to use discovery client to locate the Config Server, you can do that by setting **`spring.cloud.config.discovery.enabled: true`** (default "false").
  - The net result of that is that client apps all need a bootstrap.yml (or an environment variable) with the appropriate discovery configuration.
  - The price for using this option is an extra network round trip on start up to locate the service registration.
  - The benefit is that the Config Server can change its co-ordinates, as long as the Discovery Service is a fixed point.
- The default service id is "configserver" but you can change that on the client with **`spring.cloud.config.discovery.serviceId`**

Centralized Configuration

- To run a configuration server you need to
  - Add **`spring-cloud-config-server`** dependency
  - Add **`@EnableConfigServer`** annotation to configuration class
  - Configure property **`spring.cloud.config.server.git.uri`** to a git repo
    - for testing purposes a local repo can be used with
      - **`file://`** prefix
    - Configuration is accessible by URL
      - **`http://localhost:XXXX/AppName/profile`**



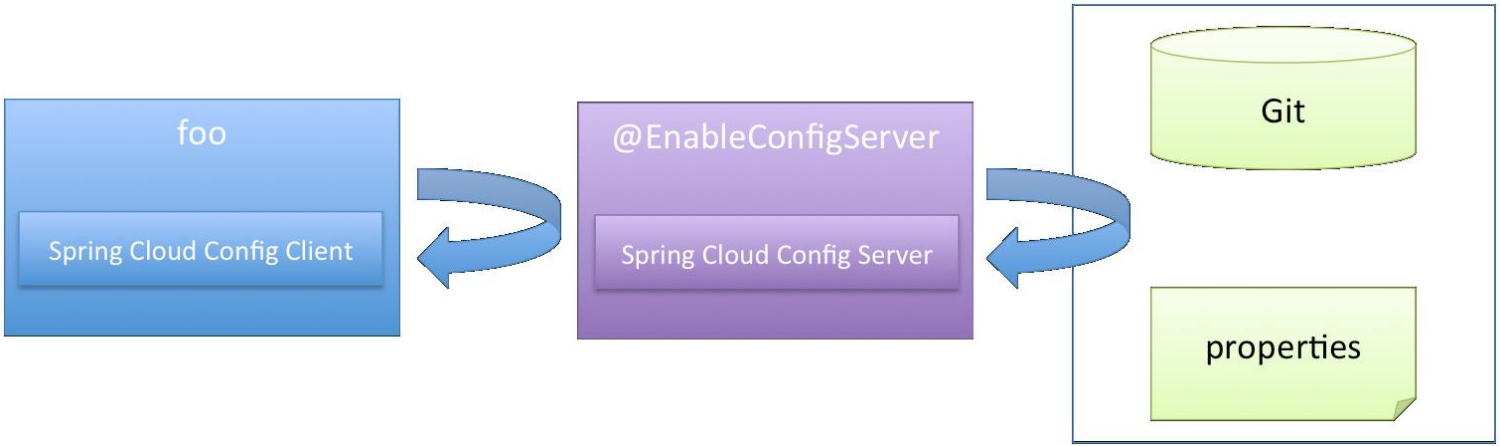
Centralized Configuration

- The configuration repository structure may contain
  - **`application.yml`** or properties file for common properties that are available to all applications;
  - a YAML or properties file with application name for a single application;
  - **`application-profile.yml`** file name format is used for profiles;
  - common properties are overridden with application specific properties;

Lab 4

Lab 4 – Centralized Configuration

- In this lab we will
  - create a configuration server with Git repository;
  - reconfigure all application to consume configuration from server;



Thank You!

LXFT
LISTED
NYSE

LUXOFT