Add instructor notes here.

Objective

- How to implement client-side load balancing with Ribbon
- How to implement a Naming Server (Eureka Naming Server)
- How to connect the micro services with the Naming Server and Ribbon
- Fault tolerance & Resilience
- Circuit Breaker Pattern-**Hystrix**

What is Eureka?

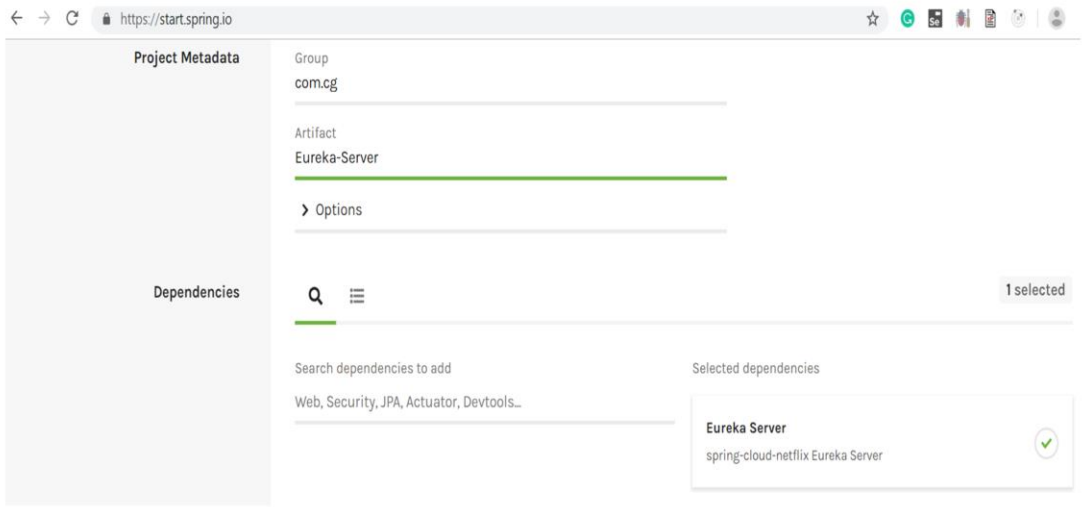
- Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.
- Eureka also comes with a Java-based client component, the **Eureka Client**, which makes interactions with the service much easier.
- 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.

Netflix Eureka

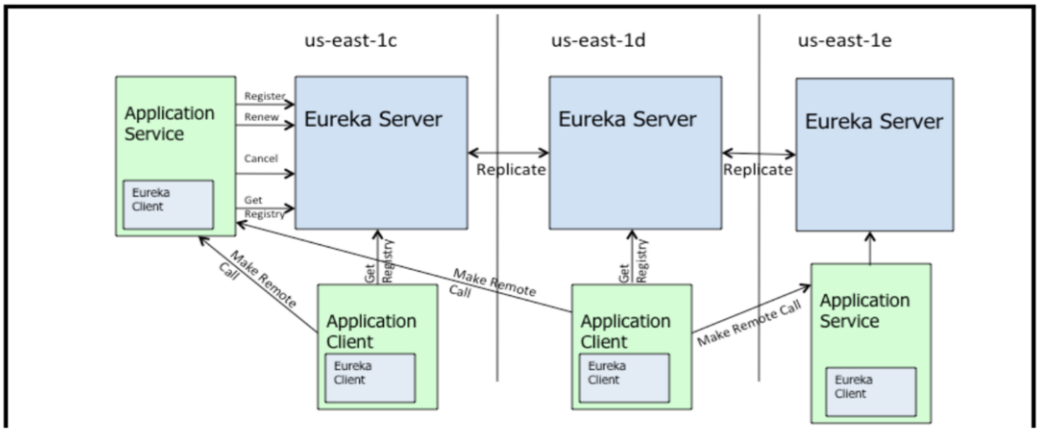
- **Netflix Eureka** is a lookup server (also called a registry). All the microservices in the cluster register themselves to this server.
- When making a REST call to another service, instead of providing a hostname and port, they just provide the service name.
- The actual routing is done at runtime along with equally distributing the load among the end services. There are other service discovery clients like Consul, Zookeeper etc

- **Eureka Server**: acts as a service registry.
- **Product Service**: a simple REST service that provides product information.
- **Recommendation Service**: a simple REST service but it internally calls the product Service to complete its requests.

Eureka-Server



Eureka-Server



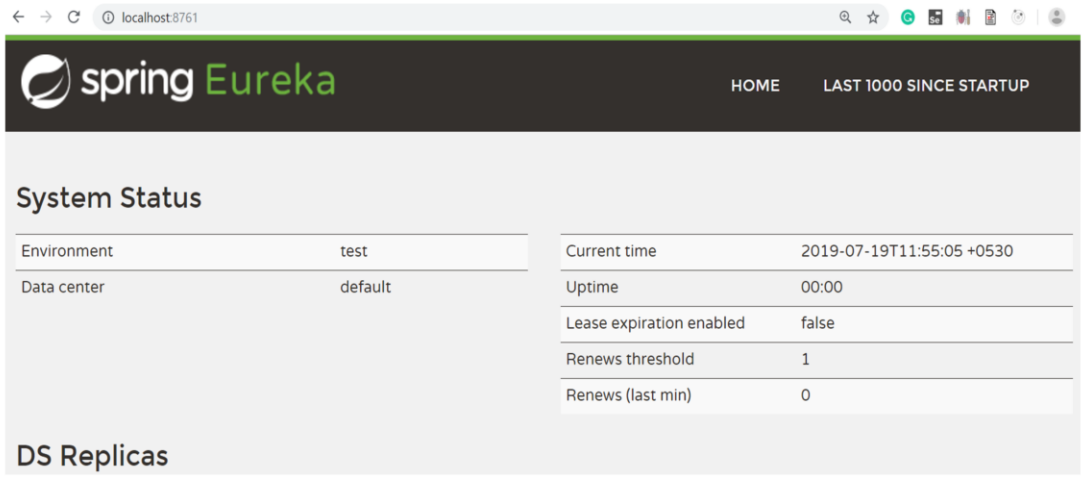
Eureka-Server

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

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

}
```

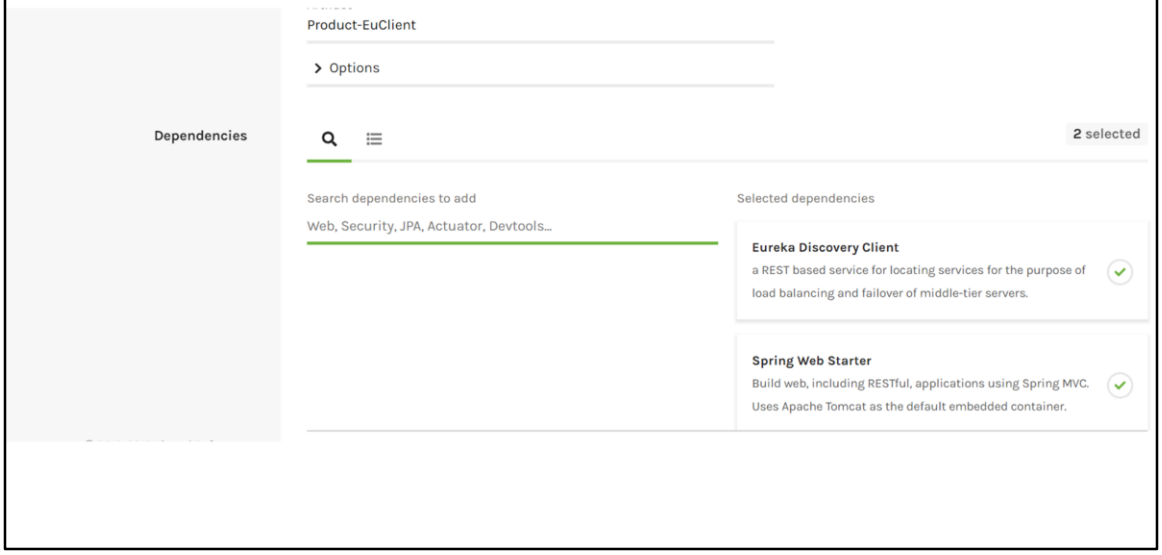
Eureka-Server-Properties

```
server.port=8761  //We can change port number

spring.application.name=eureka-server
#server.port=8761---default  port

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Eureka-Server-UP



Eureka-Client

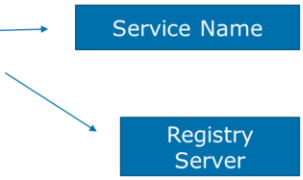


Eureka-Client

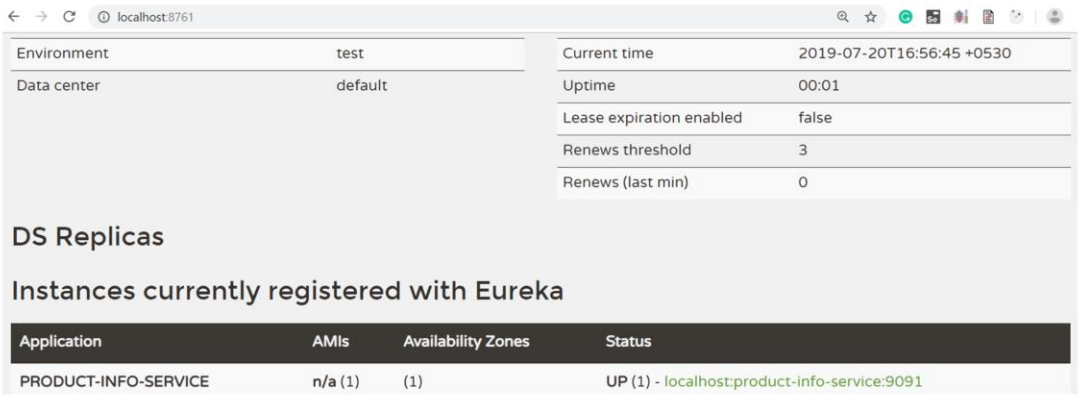
```
@SpringBootApplication
@EnableDiscoveryClient
public class ProductInfoClientApplication {

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

}
```

Eureka-Client

server.port=9091
spring.application.name=product-info-service
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka



Service Discovery -Eureka



Accessing Service from Rest Template

- We also need to create a RestTemplate bean and mark it as @LoadBalanced.
- By Ribbon we want to take advantage of client-side load balancing.
- Ribbon was developed by Netflix and later open sourced. It's dependency automatically comes with the Eureka Discovery dependency. It automatically integrates with Spring and distributes loads based on server health, performance, region, etc.
- We won't be required to use Ribbon directly as it automatically integrates RestTemplate, Zuul, Feign, etc. Using @LoadBalanced we made RestTemplate ribbon aware

- To call Client

resttemplate.getForObject("http://product-info-service/info/list", Product[].**class);**

Client Service Setup

```
@SpringBootApplication
@EnableDiscoveryClient
@ComponentScan("com.cg.productinfofront")
public class ProductinfofrontApplication {

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

}

}
```

Demo

Productinfoclient
Productinfofront
Eurekaserver

Fault tolerance & Resilience

- If Service A calls Service B, which in turn calls Service C, what happens when Service B is down? What is our fallback plan in such a scenario?

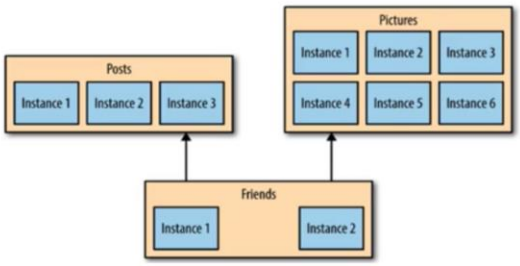
1. Can you return a pre-decided error message to the user?
2. Can you call another service to fetch the information?
3. Can you return values from cache instead?
4. Can you return a default value?

- There are a number of things we can do to make sure that the entire chain of microservices does not fail with the failure of a single component.
- There are a number of moving components in a microservice architecture, hence it has more points of failures. Failures can be caused by a variety of reasons – errors and exceptions in code, release of new code, bad deployments, hardware failures, datacenter failure, poor architecture, lack of unit tests, communication over the unreliable network, dependent services, etc.

In a monolithic application, a single error has the potential of bringing down the entire application. This can be avoided in a microservice architecture since it contains smaller independently deployable units, which won't effect the entire system. But does that mean a microservice architecture is resilient to failures? No, not at all. Converting your monolith into microservices does not resolve the issues automatically. In fact, working with a distributed system has its own challenges! While architecting distributed cloud applications, you should assume that failures will happen and design your applications for resiliency. A microservice ecosystem is going to fail at some point or another and hence you need to embrace failures. Don't design systems with the assumption that its going to go smoothly throughout the year. Be realistic and account for the chances of having rain, snow, thunderstorms, and other adverse conditions (if I may be metaphorical). In short, design your microservices with failure in mind. Things don't always go according plan and you need to be prepared for the worst case scenario

Resilience

- The problem of one component failure which can be isolated and the rest of the system can carry on working. This is a key concept in resilience engineering.
- With smaller services, we can only scale services that need scaling.
  - This way other parts of the system can still run on less powerful hardware.
- When embracing on-demand provisioning systems like those provided by Amazon Web Services, Azure, we can apply this scaling on demand for the component that need it.



Why Do We Need to Make Services Resilient?

- A problem with distributed applications is that they communicate over a network – which is unreliable. Hence we need to design our microservices so that they are fault tolerant and handle failures gracefully. In our microservice architecture, there might be a dozen of services talking with each other. We need to ensure that one failed service does not bring down the entire architecture.

How to Make our Services Resilient?

- Identify Failure Scenarios
- Avoid Cascading Failures
- Avoid Single Points of Failure
- Handle Failures Gracefully and Allow for Fast Degradation
- Design for Failures

Identify Failure Scenarios

Before releasing your new microservice to production, make sure you have tested it good enough. Strange things might happen though and you should be ready for the worst case scenario. This means you should prepare to recover from all sort of failures gracefully and in a short duration of time. This gives confidence on the system's ability to withstand failures and recover quickly with minimal impact. Hence it is important to identify failure scenarios in your architecture.

One way to achieve this is by making your microservices to fail and then try to recover from the failure. This process is commonly termed as Chaos Testing. Think about scenarios like below and find out how the system behaves:

Service A is not able to communicate with Service B.

Database is not accessible.

Your application is not able to connect to the file system.

Server is down or not responding.

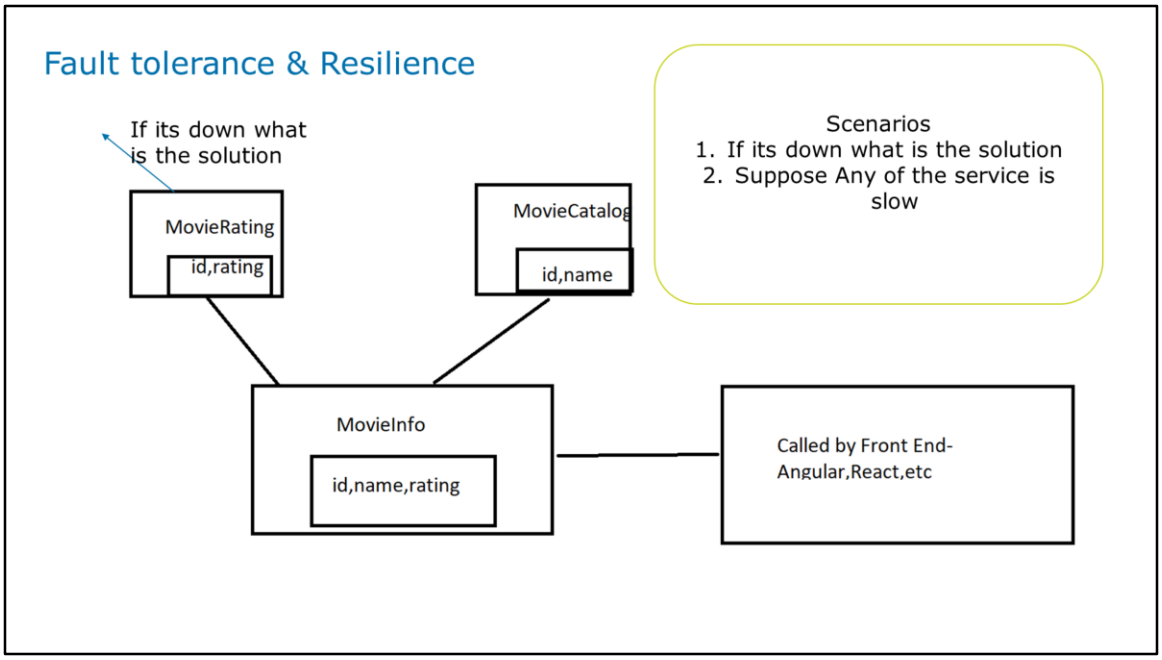
Inject faults/delays into the services.

Avoid Cascading Failures

When you have service dependencies built inside your architecture, you need to ensure that one failed service does not cause ripple effect among the entire chain. By avoiding cascading failures, you will be able to save network resources, make optimal

use of threads and also allow the failed service to recover!

Fault tolerance & Resilience



What Are the Design Patterns to Ensure Service Resiliency?

- Circuit Breaker Pattern
- Retry Design Pattern
- Timeout Design Pattern

f there are failures in your microservices ecosystem, then you need to fail fast by opening the circuit. This ensures that no additional calls are made to the failing service, once the circuit breaker is open. So we return an exception immediately. This pattern also monitor the system for failures and once things are back to normal, the circuit is closed to allow normal functionality.
This is a very common pattern to avoid cascading failure in your microservice ecosystem.
You can use some popular third-party libraries to implement circuit breaking in your application, such as Polly and Hystrix.

Retry Design Pattern

This pattern states that you can retry a connection automatically which has failed earlier due to an exception. This is very handy in case of temporary issues with one of your services. A lot of times a simple retry might fix the issue. The load balancer might point you to a different healthy server on the retry, and your call might be a success.

Timeout Design Pattern

This pattern states that you should not wait for a service response for an indefinite amount of time — throw an exception instead of waiting too long. This will ensure that you are not stuck in a state of limbo, continuing to consume application

resources. Once the timeout period is met, the thread is freed up.

Circuit Breaker Pattern

- Circuit breaker is a design pattern used in modern software development. It is used to detect failures and encapsulates the logic of preventing a failure from constantly recurring, during maintenance, temporary external system failure or unexpected system difficulties.
- Netflix's Hystrix library provides an implementation of the circuit breaker pattern. When you apply a circuit breaker to a method, Hystrix watches for failing calls to that method, and, if failures build up to a threshold, Hystrix opens the circuit so that subsequent calls automatically fail. While the circuit is open, Hystrix redirects calls to the method, and they are passed to your specified fallback method.

Context and problem

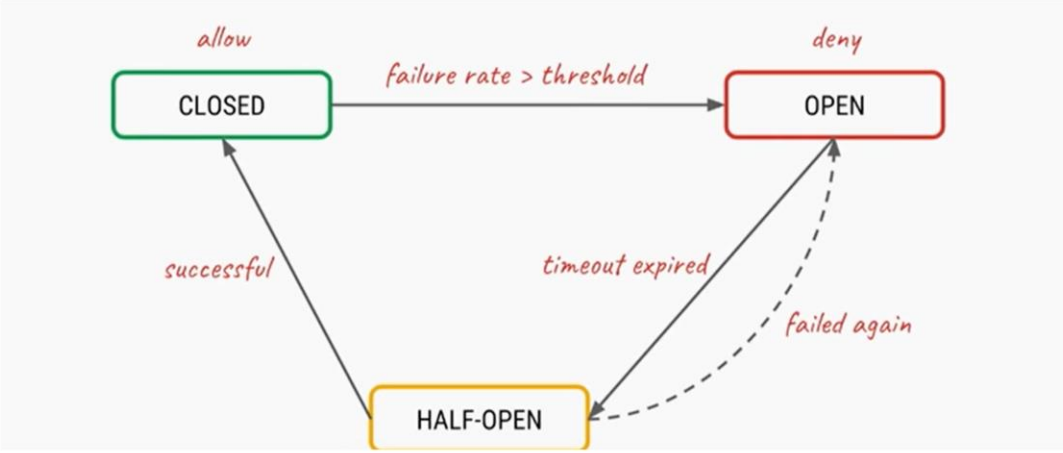
In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them by using a strategy such as the Retry pattern.

However, there can also be situations where faults are due to unanticipated events, and that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure in one part of the system might lead to cascading failures. For example, an operation that invokes a service could be configured to implement a timeout, and reply with a failure message if the service fails to respond within this period. However, this strategy could cause many concurrent requests to the same operation to be blocked until the timeout period expires. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on. Consequently, these resources could

become exhausted, causing failure of other possibly unrelated parts of the system that need to use the same resources. In these situations, it would be preferable for the operation to fail immediately, and only attempt to invoke the service if it's likely to succeed. Note that setting a shorter timeout might help to resolve this problem, but the timeout shouldn't be so short that the operation fails most of the time, even if the request to the service would eventually succeed.

Circuit Breaker Pattern



Solution

The Circuit Breaker pattern, popularized by Michael Nygard in his book, Release It!, can prevent an application from repeatedly trying to execute an operation that's likely to fail. Allowing it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long lasting. The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.
The purpose of the Circuit Breaker pattern is different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it'll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.
A circuit breaker acts as a proxy for operations that might fail. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.

What Is Hystrix?

- 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.

What Is Hystrix For?

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.

What Problem Does Hystrix Solve?

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.

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.

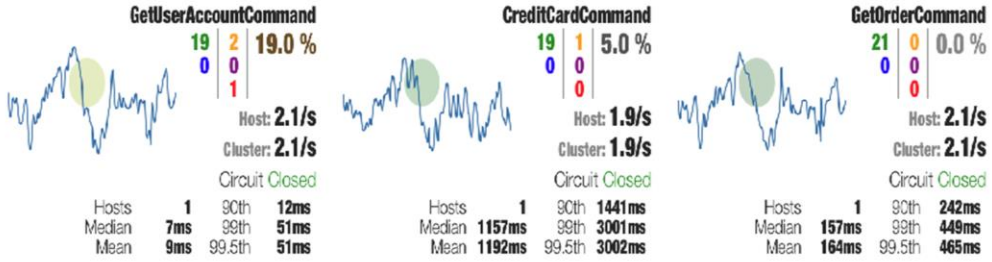
Reality is generally worse.

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**.

Hystrix Dashboard

The Hystrix Dashboard allows us to monitor Hystrix metrics in real time.

When Netflix began to use this dashboard, their operations improved by reducing the time needed to discover and recover from operational events. The duration of most production incidents (already less frequent due to Hystrix) became far shorter, with diminished impact, due to the real-time insights into system behavior provided by the Hystrix Dashboard.

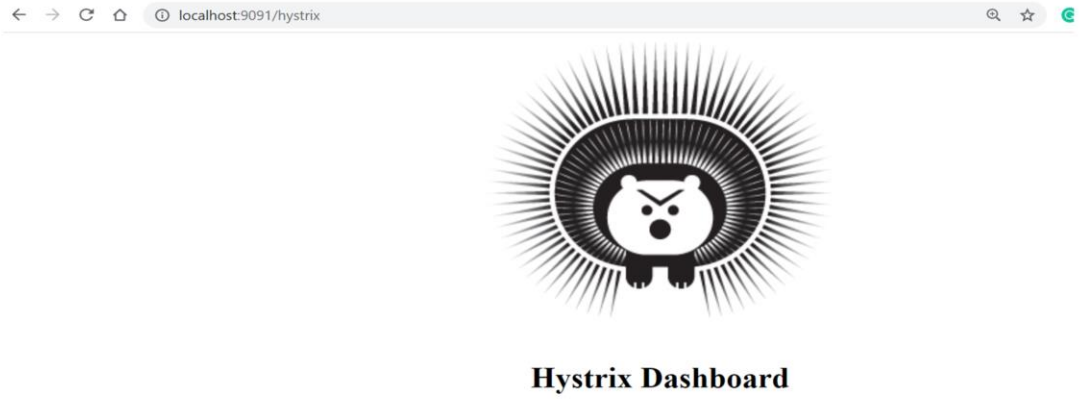


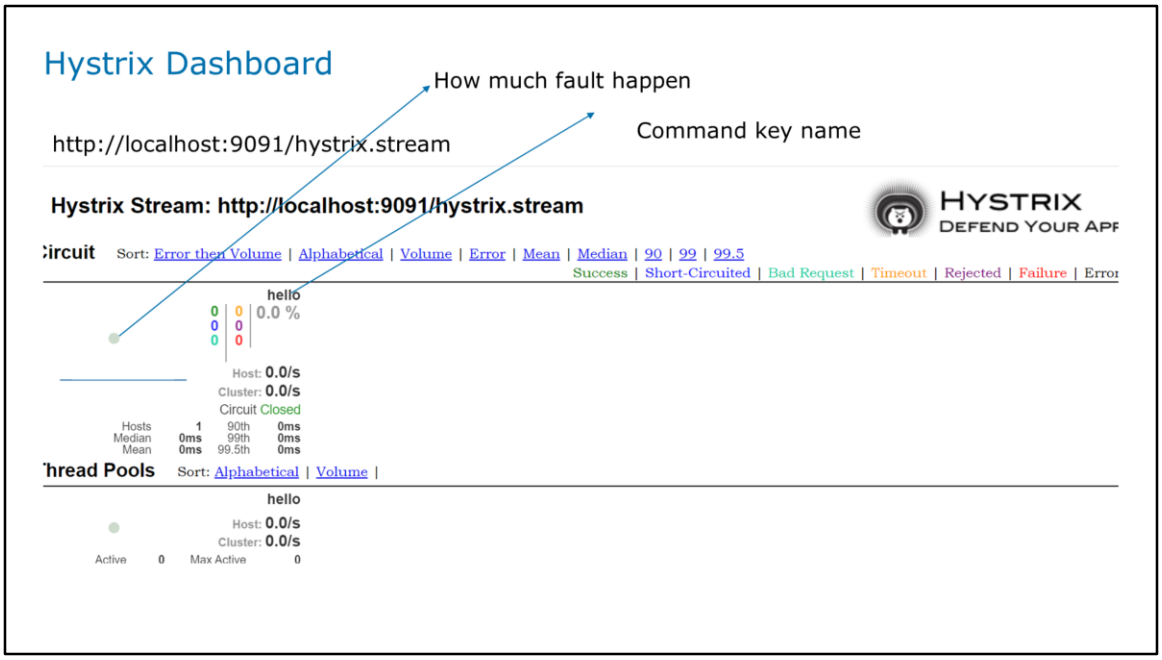
Hystrix Dependency

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
```

Hystrix Dashboard

http://localhost:9091/hystrix




Hystrix Dashboard
How much fault happen
Command key name
http://localhost:9091/hystrix.stream
Hystrix Stream: http://localhost:9091/hystrix.stream
HYSTRIX
DEFEND YOUR APF
ircuit
Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5
Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Erro
hello
0 0 0.0 %
0 0
0 0
Host: 0.0/s
Cluster: 0.0/s
Circuit Closed
Hosts 1 90th 0ms
Median 0ms 99th 0ms
Mean 0ms 99.5th 0ms
hread Pools
Sort: Alphabetical | Volume |
hello
Host: 0.0/s
Cluster: 0.0/s
Active 0 Max Active 0

Demo

Demohystrix

Lab

Lab 2