Microservice Architecture Blueprint

Episode 5 "Avoiding Icebergs"

Kim Horn

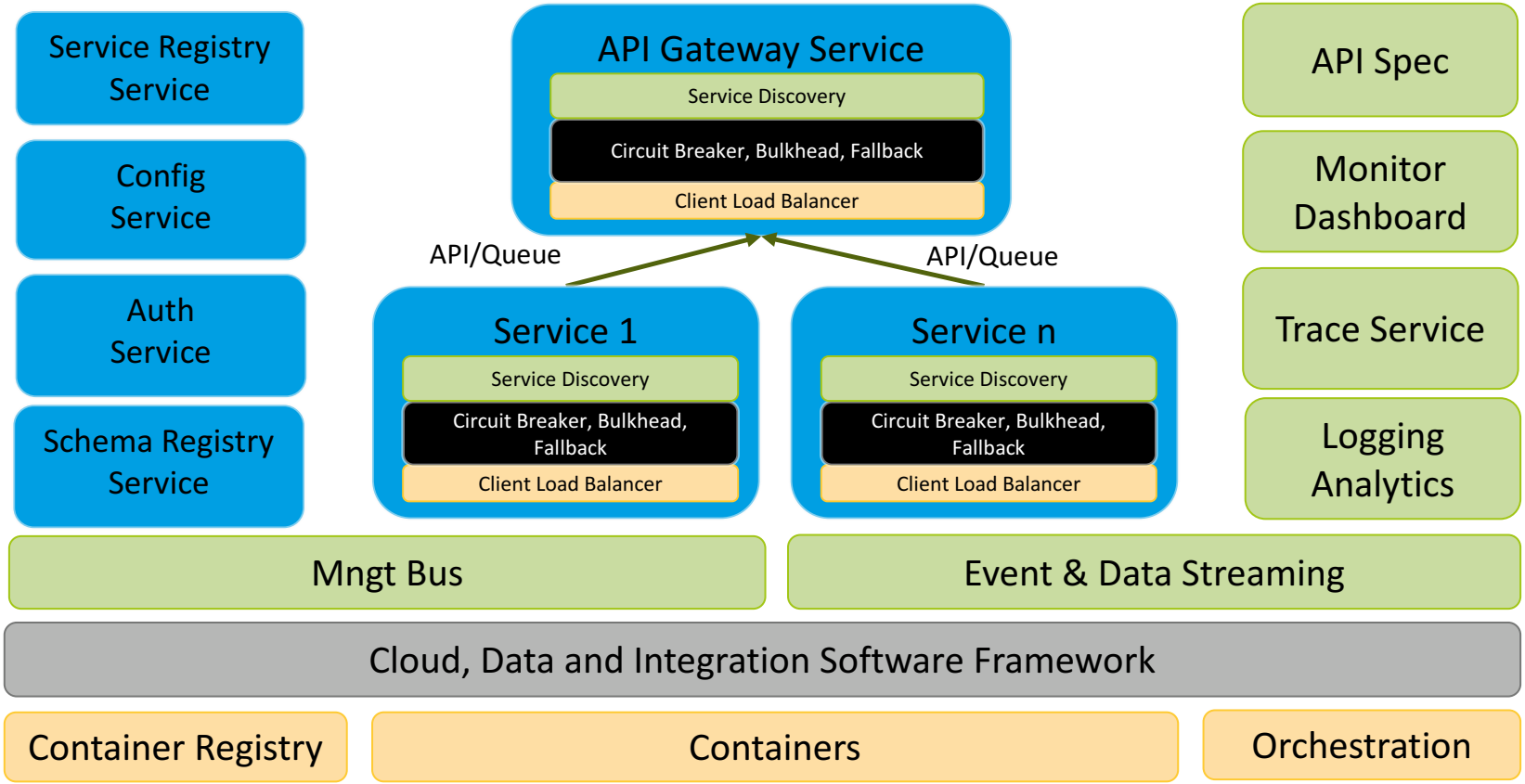
Version 1.32 **1 April 2018**

Agenda

- Handling Failures – design for Failure
- Service Health
- Client/Server Side Load Balancer
- Hystrix:
  - Circuit Breaker
  - Bulkhead
  - Fallback
- Monitoring : Hystrix Dashboard, Turbine
- Zuul Revisited – an Integrated Cohesive Service
- Distributed Tracing and Logging : Sleuth, Zipkin, logspout, ELK
- Demo

Working Blueprint

Distributed Cloud Blueprint – NFRs, use what you need



Common Tech For Patterns and Stack

Pattern / Capability	Tech
Gateway (reverse proxy)	Zuul 1&2 (Netflix) – wrapped by SpringCloud, Spring Cloud Gateway
Circuit Breaker, Bulkhead, Fallback	Hysterix (Netflix) – wrapped by SpringCloud
Client Side Load Balancer	Ribbon (Netflix) – wrapped by SpringCloud
Service Discovery Registry	Eureka (Netflix) – wrapped by SpringCloud / Consul
REST HTTP and Service Discovery Client	Feign (Netflix) – wrapped by SpringCloud
Configuration Service	Spring Cloud Configuration
Authorisation Service	Spring Cloud Security, SAML, OATH, OpenID
Event Driven Architecture, Mngt Bus	Spring Cloud Stream wraps Kafka (also supports Rabbit MQ), Spring Cloud Bus
Data Analysis and Flow Pipelines	Spring Cloud Data, Spring Cloud Data Flow, Sparks
Core Software Cloud Frameworks	Spring Boot, Spring Actuator, Spring Cloud, Spring Data, Spring Integration
IaaS, Container, Orchestration	Docker, Compose [Kubernites, Swarm]
Logging, Tracing & Analysis	SL4J, ELK[Elastisearch, Logstash, Kibana], Sprint Cloud Sleuth, Spring Cloud Zipkin.
In memory Cache	Spring Data Redis
Analytics, Monitoring Dashboard, Orchestration(bpm)	Atlas, Turbine, Visceral, Camuda, Conductor (lightweight)
API/ Service Specs, Schema Registry,Testing	Spring Hateoas (HAL), Spring Fox (Swagger), Avro, Protobuf, Spring Cloud Contract, WireMock

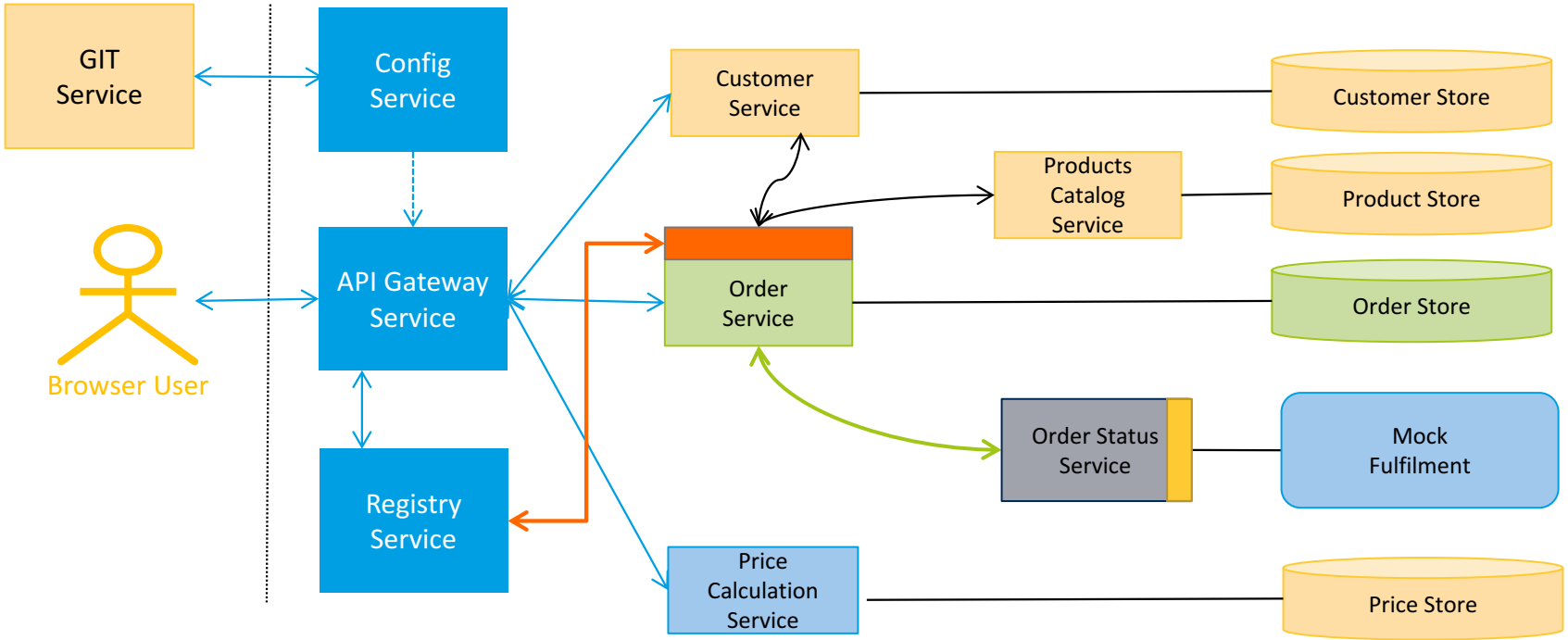
Microservice Principles

A Microservice is:

1. Modelled around a bounded context; the business domain.
2. Responsible for a single capability;
3. Based on a culture of automation and continuous delivery;
4. The owner of its data;
5. Consumer Driven;
6. Not open for inspection; Encapsulates and hides all its detail;
7. Easily observed;
8. Easily built, operated, managed and replaced by a small autonomous team;
9. A good citizen within their ecosystem;
10. Based on Decentralisation and Isolation of failure;

Recap - Discovery Clients

RP



Pattern: Client Discovery

Providing Qualities

- **Scalability, Elasticity** - Autoscaling:
  - Scale up to meet load or performance;
  - Scale down to reduce costs;
- **Reliability** – reduce manifest service failures and cascading effects.
  - Flag unhealthy instances and remove from service.
  - Employ reliability patterns to manage network issues;
- **Availability** – when services fail or become unhealthy, fail fast, and quickly replace; make them available for use (registry).

✓ To provide these qualities a cohesive framework is required.

✓ To provide agility the framework needs to reduce the time to develop a rich set of proven capabilities.

✓ To effectively manage and operate the application, monitoring and logging capabilities are essential that work across services;

Service Health

Insidious Problems

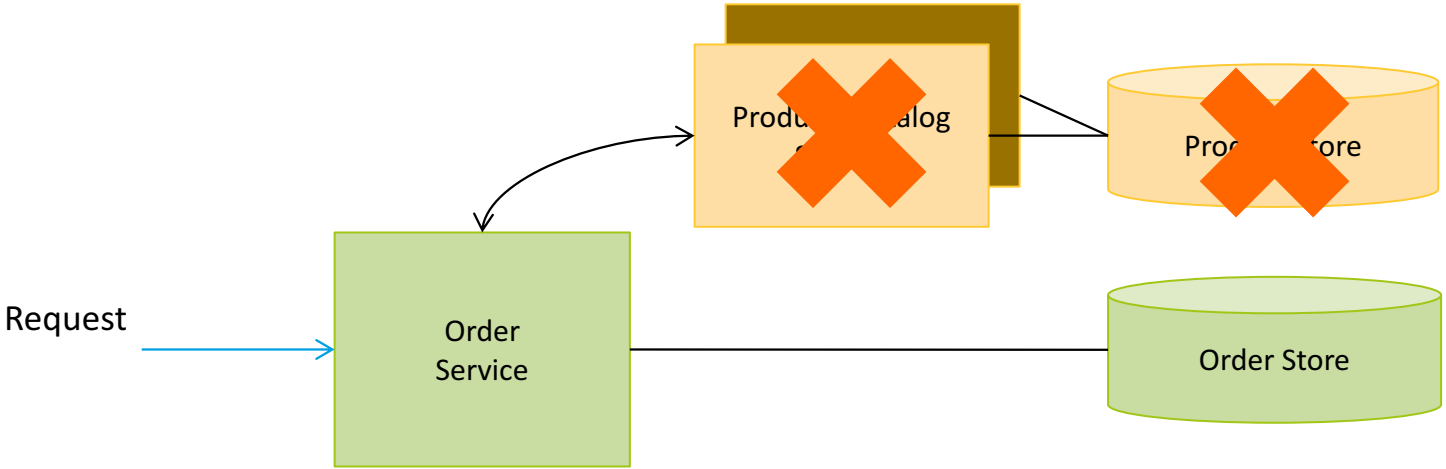
Most applications don't deal well with:

- Faults that are intermittent and that slowly build momentum;
- Synchronous calls to remote services, and often do not cut short long running calls;
- Partial degradations of remote resources, despite dealing with complete failures;
  - They will continue to call a poorly behaving service.
- Resource exhaustion that can result is when a limited resource like a thread pool or database connection "maxes" out, and the calling client must wait for that resource to become available;
- Alternate strategies that are designed to stop the cascading effects.

Micro services need to handle all these issues.

Handling Failures

CP



- What happens to Order Service when the Product Store DB or Product Catalog Service is down ?
- How can we Load Balance when we need to scale up the Product Catalog; Hardware or Software ?
- What happens when service instances are not well, poorly performing ?

Core Client Side Resiliency Patterns

These patterns are implemented in the client calling a remote resource:

- Timeouts
- Retries
- Client Side Load Balancing
  - Cache Instances locally (resiliency)

Ribbon

- Circuit Breaker
- BulkHead
- FallBack

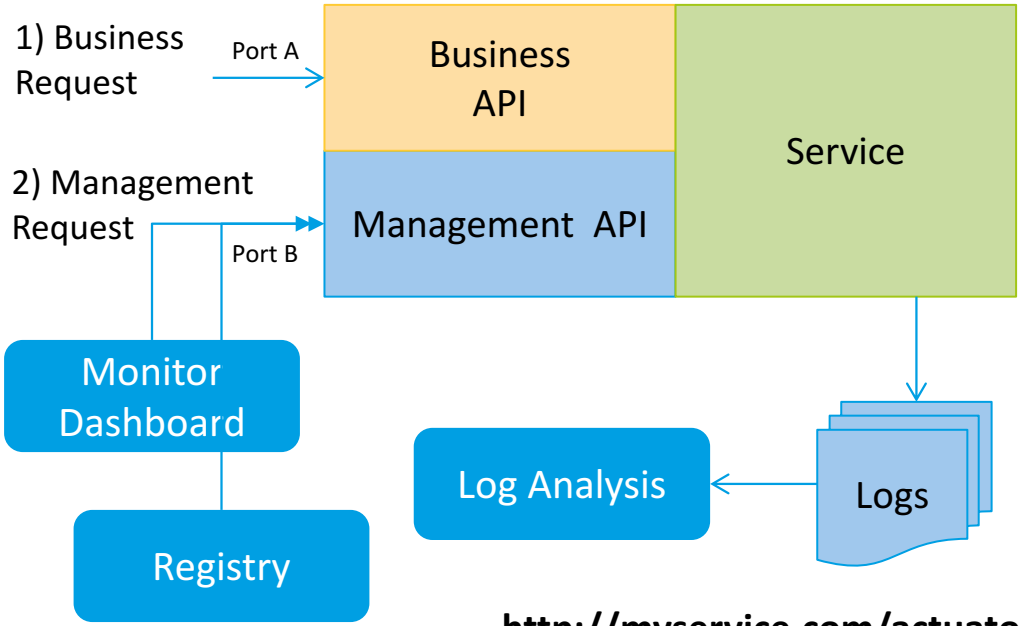
Hystrix

We may also need:

- Throttling – to protect downstream services (Zuul)
- Monitoring – across the stack (e.g. ELK, Zipkin)

Service Health : Management APIs

- Ask a service about its health;
- Used for routing and load balancing;
- **Two main APIs for a Service:**



http://myservice.com/actuator/health


```
{
 "description": "Spring Cloud Eureka Discovery Client",
 "status": "UP",
 "discoveryComposite": {
  "description": "Spring Cloud Eureka Discovery Client",
  "status": "UP",
  "discoveryClient": {
   "description": "Spring Cloud Eureka Discovery Client",
   "status": "UP",
          },
  "eureka": {
   "description": "Remote status from Eureka server",
   "status": "UP",
          }
 },
 "diskSpace": {
  "status": "UP",
  "total": 63375708160,
  "free": 59655077888,
  "threshold": 10485760
          },
 "refreshScope": {  "status": "UP"  },
 "hystrix": {  "status": "UP"  }
}
```


http://myservice.com/actuator/mappings

Spring Actuator

Actuator provides a default set of monitoring and management endpoints.

Features include:

- **Endpoint** via HTTP, JMX or SSH;
- **Metrics** service: “gauge” and “counter”. A “gauge” records a single value; and a “counter” records a delta (an increment or decrement). Metrics for all HTTP requests are automatically recorded;
- **Audit** framework that will publish events to an AuditEventRepository. Using Spring Security it automatically publishes authentication events by default, for reporting, and to implement a lock-out policy based on authentication failures;
- **Process Monitoring** you can find ApplicationPidFileWriter which creates a file containing the application PID (by default in the application directory with a file name of application.pid);
- To provide security it can be opened on its own port by setting *‘management.port’;*
- The Business API can be set separately with *‘server.port’.*

http://localhost:8080/actuator


```
{  "_links": {
"self": {  "href": "http://localhost:8080/actuator",
"templated": false  },
   "archaius": {  "href": "http://localhost:8080/actuator/archaius",
"templated": false  },
 "auditevents": {  "href": "http://localhost:8080/actuator/auditevents",
"templated": false  },
"beans": {  "href": "http://localhost:8080/actuator/beans",  "templated":
false  },
"health": {  "href": "http://localhost:8080/actuator/health",
"templated": false  },
"conditions": {  "href": "http://localhost:8080/actuator/conditions",
"templated": false  },
"configprops": {  "href": "http://localhost:8080/actuator/configprops",
"templated": false  },
"env-toMatch": {  "href": "http://localhost:8080/actuator/env/{toMatch}",
"templated": true  },
"env": {  "href": "http://localhost:8080/actuator/env",  "templated":
false  },
 "info": {  "href": "http://localhost:8080/actuator/info",  "templated":
false  },
"loggers": {  "href": "http://localhost:8080/actuator/loggers",
"templated": false  },
"loggers-name": {  "href": "http://localhost:8080/actuator/loggers/{name}",
"templated": true  },
"heapdump": {  "href": "http://localhost:8080/actuator/heapdump",
"templated": false  },
"threaddump": {  "href": "http://localhost:8080/actuator/threaddump",
"templated": false  },
"metrics-requiredMetricName": {  "href":
"http://localhost:8080/actuator/metrics/{requiredMetricName}",
"templated": true  },
"metrics": {  "href": "http://localhost:8080/actuator/metrics",
"templated": false  },
"scheduledtasks": {  "href": "http://localhost:8080/actuator/scheduledtasks",
"templated": false  },  "httptrace": {  "href":
"http://localhost:8080/actuator/httptrace",  "templated": false  },
"mappings": {  "href": "http://localhost:8080/actuator/mappings",
"templated": false  },  "refresh": {  "href":
"http://localhost:8080/actuator/refresh",  "templated": false  },
"features": {  "href": "http://localhost:8080/actuator/features",
"templated": false  },  "service-registry": {  "href":
"http://localhost:8080/actuator/service-registry",  "templated": false  },
"hystrix.stream": {  "href": "http://localhost:8080/actuator/hystrix.stream",
"templated": false  }  }}
```

Spring Boot 1 : Metrics - http://localhost:8081/metrics

Over 148 lines of metrics by default……


```
{
"mem": 1091780,
"mem.free": 130772,
"processors": 6,
"instance.uptime": 1310748,
"uptime": 1407574,
"systemload.average": 0.15576171875,
"heap.committed": 990208,
"heap.init": 159744,
"heap.used": 859435,
"heap": 2272256,
"nonheap.committed": 103168,
"nonheap.init": 2496,
"nonheap.used": 101573,
"nonheap": 0,
"threads.peak": 45,
"threads.daemon": 43,
"threads.totalStarted": 54,
"threads": 45,
……
……
……
……

"counter.servo.timeouts": 0,
"counter.servo.rejectedexecutions": 0,
"counter.servo.throwables": 0,

……
……

gauge.servo.eurekaclient.registry.lastupdatesec_00240": 0,
"gauge.servo.hystrix.hystrixcommand.productservice.getprod.rollingcountfallbackmissing": 0,
"gauge.servo.hystrix.hystrixthreadpool.productservice.propertyvalue_queuesizerejectionthreshold": 5,
"httpsessions.max": -1,
"httpsessions.active": 0,
"datasource.primary.active": 0,
"datasource.primary.usage": 0
}
```

Spring Boot 2 : Metrics - http://localhost:8081/actuator/metrics

Now it lists the metrics you can access, a second call is required for each.


```
{
"mem": 1091780,
"mem.free": 130772,
"processors": 6,
"instance.uptime": 1310748,
"uptime": 1407574,
"systemload.average": 0.15576171875,
"heap.committed": 990208,
"heap.init": 159744,
"heap.used": 859435,
"heap": 2272256,
"nonheap.committed": 103168,
"nonheap.init": 2496,
"nonheap.used": 101573,
"nonheap": 0,
"threads.peak": 45,
"threads.daemon": 43,
"threads.totalStarted": 54,
"threads": 45,
……
……
```


To Update


```
……
……

"counter.servo.timeouts": 0,
"counter.servo.rejectedexecutions": 0,
"counter.servo.throwables": 0,

……
……

gauge.servo.eurekaclient.registry.lastupdatesec_00240": 0,
"gauge.servo.hystrix.hystrixcommand.productservice.getprod.rollingcountfallbackmissing": 0,
"gauge.servo.hystrix.hystrixthreadpool.productservice.propertyvalue_queuesizerejectionthreshold": 5,
"httpsessions.max": -1,
"httpsessions.active": 0,
"datasource.primary.active": 0,
"datasource.primary.usage": 0
}
```

Eureka's Health Checks

- By default, Eureka uses the client heartbeat to determine if a client is up.
- Unless specified otherwise the Discovery Client will not propagate the current health check status of the application per the Spring Boot Actuator.
  - So after successful registration Eureka will always announce that the application is 'UP'.
  - Enabling Eureka health checks results in propagating application status to Eureka.
  - In application.yml set *eureka: client: healthcheck: enabled*: true
- *eureka.instance.leaseExpirationDurationInSeconds* needs to be tuned if you want Eureka to clean away the dead instances faster. The default value is 90.

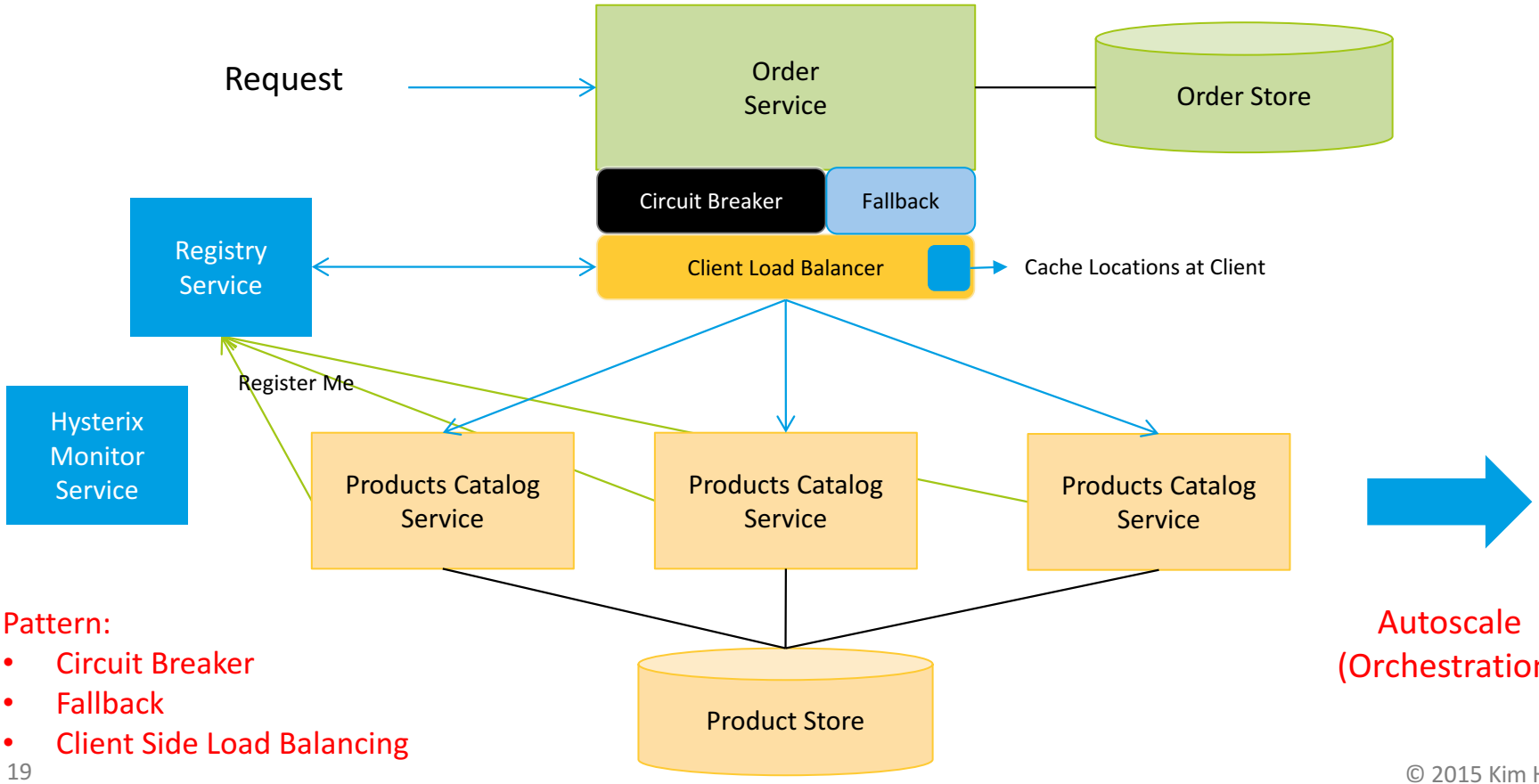
Microservices Require Load Balancing

Microservices applications require load balancing to an even greater degree than monolithic applications.

- Microservices relies on multiple, small services working in concert to provide application functionality. This inherently requires robust, intelligent load balancing, especially where external clients access the service APIs directly;
  - Add/remove microservice as required to cope with load (scale up) and reduce costs (scale down);
  - Add/remove microservice to cover sick microservice;
- In a monolithic app, the objects or functions communicate in memory and share data through pointers or object references;
- Micoservices communicate over the network, typically using HTTP. So the network is a critical bottleneck in a microservices application, as it is inherently slower than in-memory communication;
- All networking aspect become crucial and delays can have a significant impact.

Scaling - Client Load Balancer

CP



Pattern:

- Circuit Breaker
- Fallback
- Client Side Load Balancing

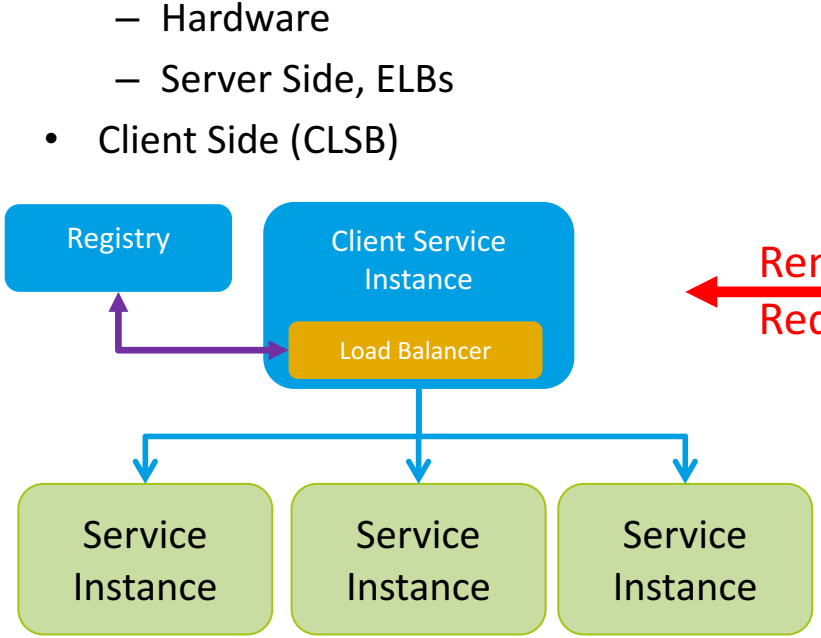
Autoscale (Orchestration)

Load Balancing – Choices

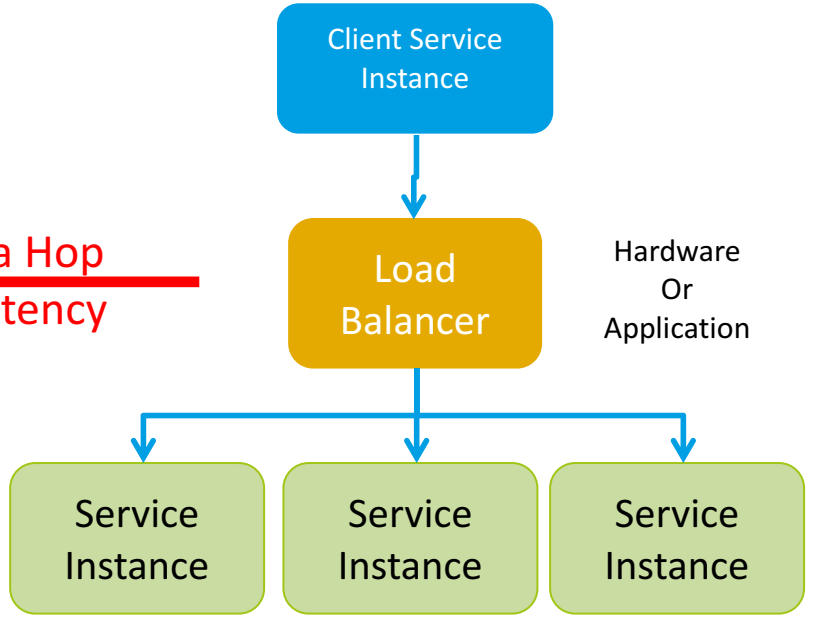
Three main choices:

- Infrastructure:
  - Hardware
  - Server Side, ELBs
- Client Side (CLSB)

Scalability requires software defined infrastructure



Removes a Hop
Reduce Latency



Isn't DNS the right way ?

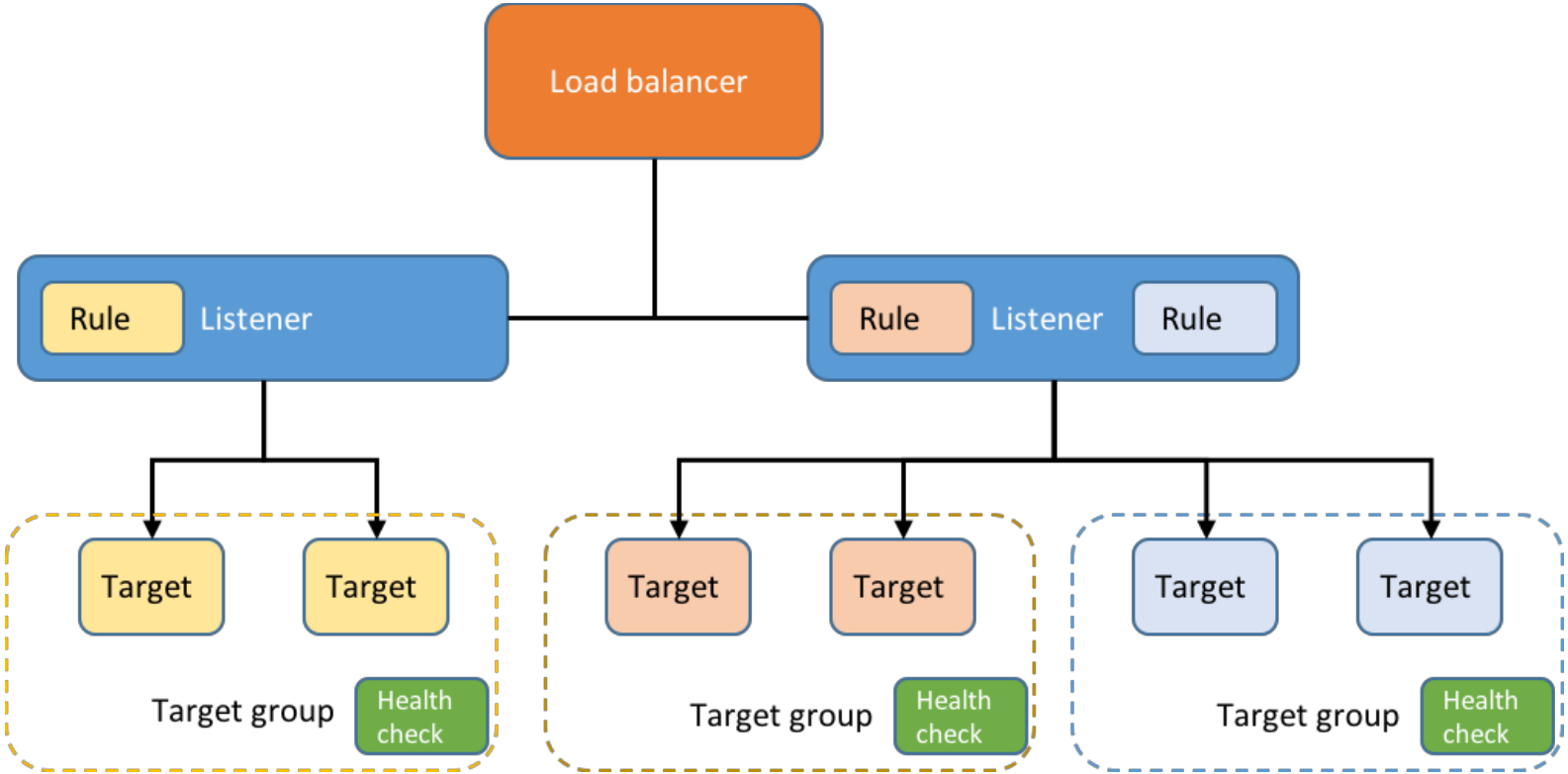
AWS Elastic Load Balancers

Elastic Load Balancing has 3 types:

- **Classic (CLB)** - Layer 4 (IP and Port) or 7 but require a fixed relationship with container port instances.
- **Network** - Layer 4 (IP and Port)
- **Application (ALB)** - Layer 7 (app level routes, e.g. URL)
  - Good fit for ECS Services;
  - Provide service discovery for its services (providers) only;
  - Allow containers to use dynamic port mapping;
  - Support path routing so multiple services can use same port on a single ALB.

- These mechanisms are independent of the software, and usually managed by separate team.
- Requires independent configuration and management to software;

AWS – Application Load Balancer



Issues with Infrastructure Approach

Infrastructure Load balancing (ELB / Hardware)

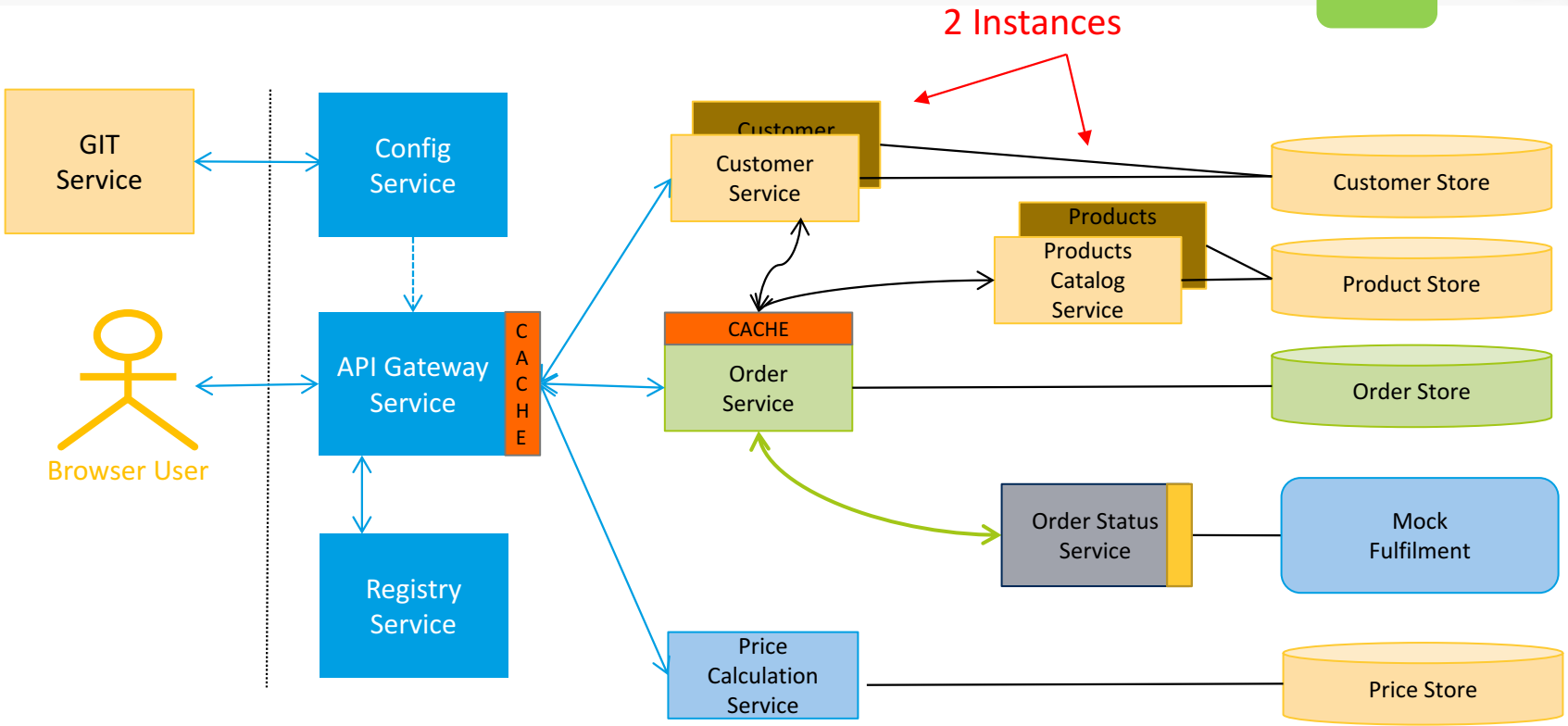
- Might be a hardware configuration or an infrastructure configuration not under software control;
- If a server stops responding have to remove this IP from the load balancer, this knowledge is not easily available to software;
- Have to failover clients seamlessly, application is not aware of wha'ts going on;
- Have to provide extra redundancy for the load balancer, as it may fail, and an extra mechanism to manage that redundancy; this all makes control and management complex;

Microservice Client Side (CLSB)

- Each client provides a redundant load balancer itself;
- Each client is resilient as it caches its own list of working providers, if registry or ELB, is not available;
- Each client can determine how it needs to load balance, its algorithm;
- Software determines the network configuration not the cabling/infrastructre configuration;
- Load balancing is under developer control and does not require independent infrastructure (cloud) team;
- Clients can decide to talk to whoever they want, and load balance as required.
- CSLB + Registry – Provides a consistent overall architecture, under developer control, that makes teams more agile; registry is overall knowledge source for the application, not localised as an ALB is. See Visceral, (later slide) big picture awareness;

Client Side Load Balancing

RP



Pattern: Client Side Load Balancing (at Gateway and Order Service), Local Cache

Load Balancer - Ribbon Discovery

A special end point, in order service I created to see the Order Service Discovery Cache:

http://localhost:8080/v1/tools/eureka/services


```
[
  "configserver:http://a5042cc176b3:8888",
  "zuulservice:http://172.18.0.4:5555",
  "customerservice:http://172.18.0.8:8085",
  "customerservice:http://172.18.0.7:8085",
  "productservice:http://172.18.0.3:8081",
  "productservice:http://172.18.0.12:8081",
  "orderservice:http://172.18.0.5:8080"
]
```

Non Spring Cloud Apps can access Registry too.

Interfacing a standard Java program (not using SideCar), for example using Java.Net:

Create an API call to registry and get instances:

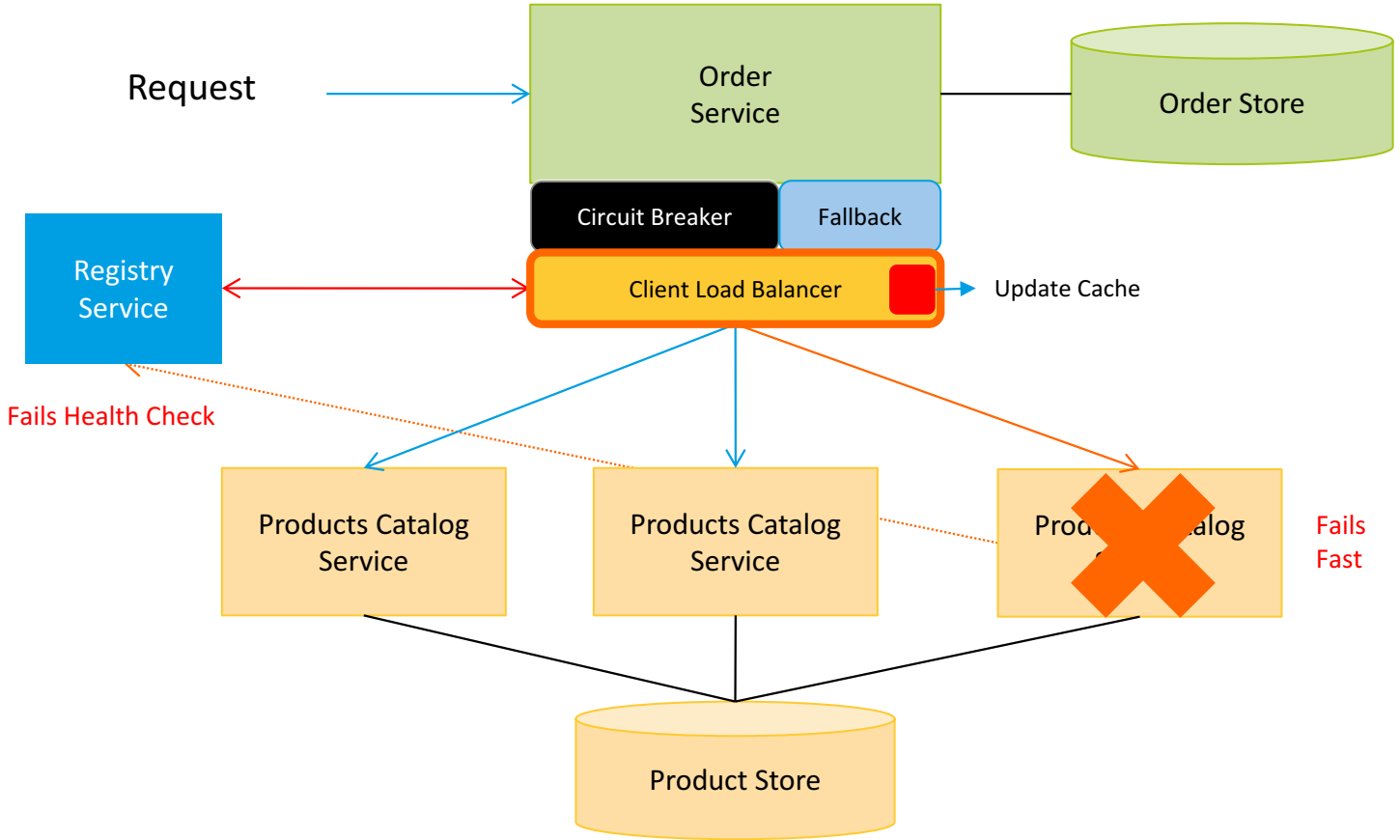
URL = “http://localhost:8761/eureka/apps/” + appname

Parse/Search the response to find the instances:

```
<application>
    <name>PRODUCTSERVICE</name>
        <instance>
            <instanceId>1e87686b2e6a:productservice:8081</instanceId>
        <hostName>172.18.0.4</hostName>
            <ipAddr>172.18.0.4</ipAddr>
            <status>UP</status>
```

Remove Unhealthy Nodes

CP



Hystrix

Hystrix – Defend You App

"Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services, and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable."
dependency chains.

- Hystrix is essential, it protects applications from cascading dependency failures, an issue common to complex distributed architectures, with multiple dependency chains.
- It uses multiple isolation techniques, such as bulkhead, swimlane, and circuit breaker patterns, to limit the impact of any one dependency on the entire system.

Hystrix

Concept

- Hystrix is a Java Library which provides fault tolerant access to external services.
- Maintains system stability, providing a responsive experience for clients.

Problems Addressed

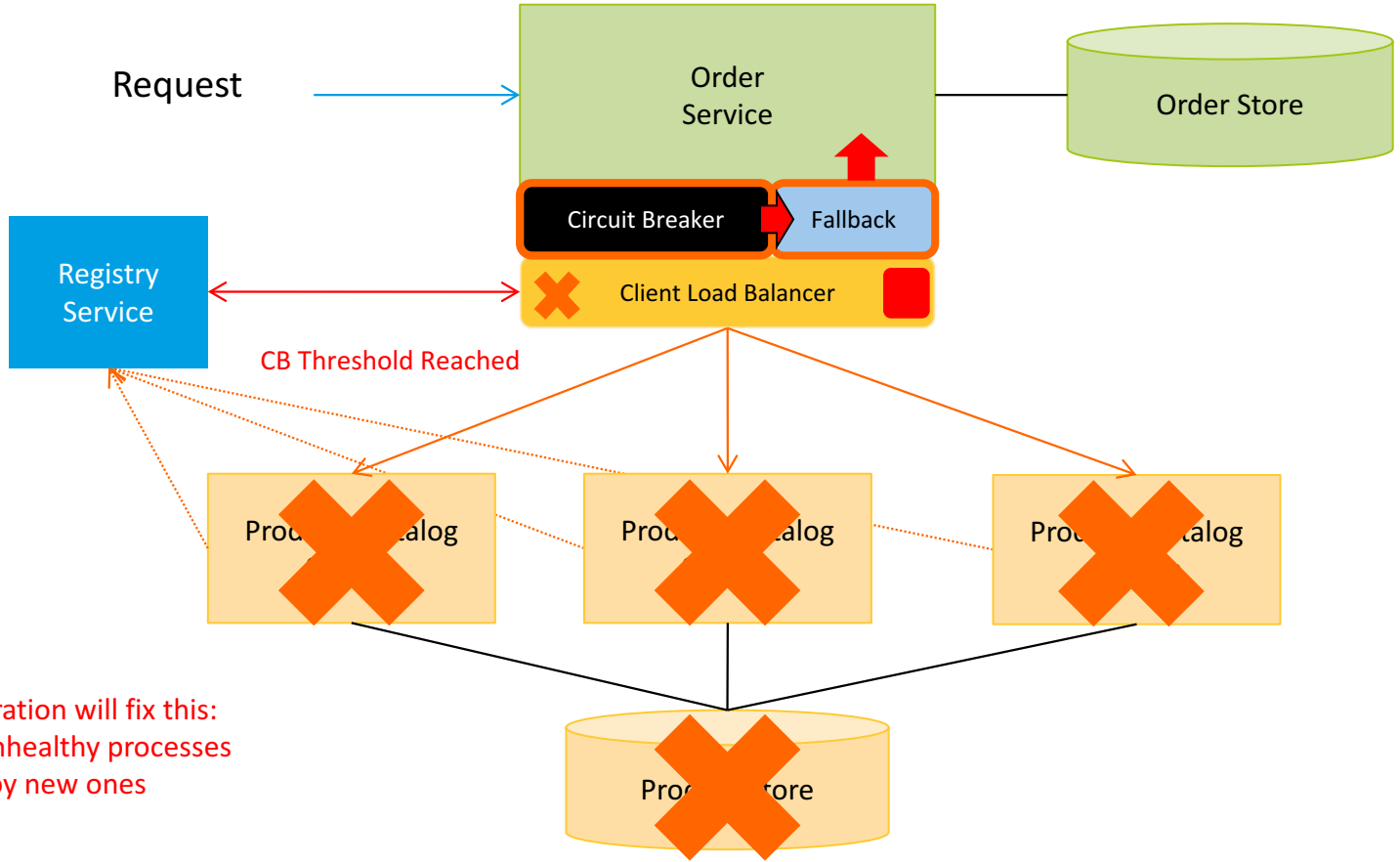
- When resources are held for longer than necessary, resources can be exhausted and systems can become unresponsive. Resources include threads, network connections, memory etc.
- When resource use is un-constrained and/or un-batched, it can slow and make systems unresponsive
- When resource use is un-constrained, failures in systems can spread, causing cascading failures across systems

Hystrix Solution

- Connections to external services are cancelled after a timeout; freeing system resources, and making systems responsive
- Thread and network usage is constrained through Thread Pools and Semaphores
- When resources are consumed up to their constraints, further requests are failed, rather than queued
  - When a failure occurs, Fallback options can be used when appropriate; making exceptional circumstances more palatable for clients
- Request Collapsing can be employed to batch requests; to more efficiently use local and external service resources
- To further constrain unnecessary resource usage, Circuit Breakers are employed at each external service.
  - When access to an external service becomes problematic, the circuit is tripped and access is immediately failed. One request is periodically allowed through, to test if connectivity can be re-established

Circuit Breaker and Fallback

CP



Orchestration will fix this:

- Kill unhealthy processes
- Deploy new ones

Littles Law and Hystrix Thread Size

Little's law says that the number of requests in a system equals the rate at which they arrive, multiplied by the average amount of time it takes to service an individual request.

Example:

- the mean response time = 100ms.
- requests per second = 200.

The mean utilization = (0.1 s) * (200 rps) = 20 requests.
As this is the mean, it will burst much higher than 20, a large number of threads.

Hystrix recommends ThreadPoolSize = requests per second at peak when healthy × 99th percentile latency in seconds + some breathing room

If 99% tile response time is ~500ms, given above, the thread pool size = 100.

The formula above is just an estimate. You need to move the number around until you get to a rejection rate you're comfortable with. A thread pool of 30 will give a a rejection rate of 4%, or increase it to 80 to get it down to 1%.

getOrderItems Fallback

Get http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a/items

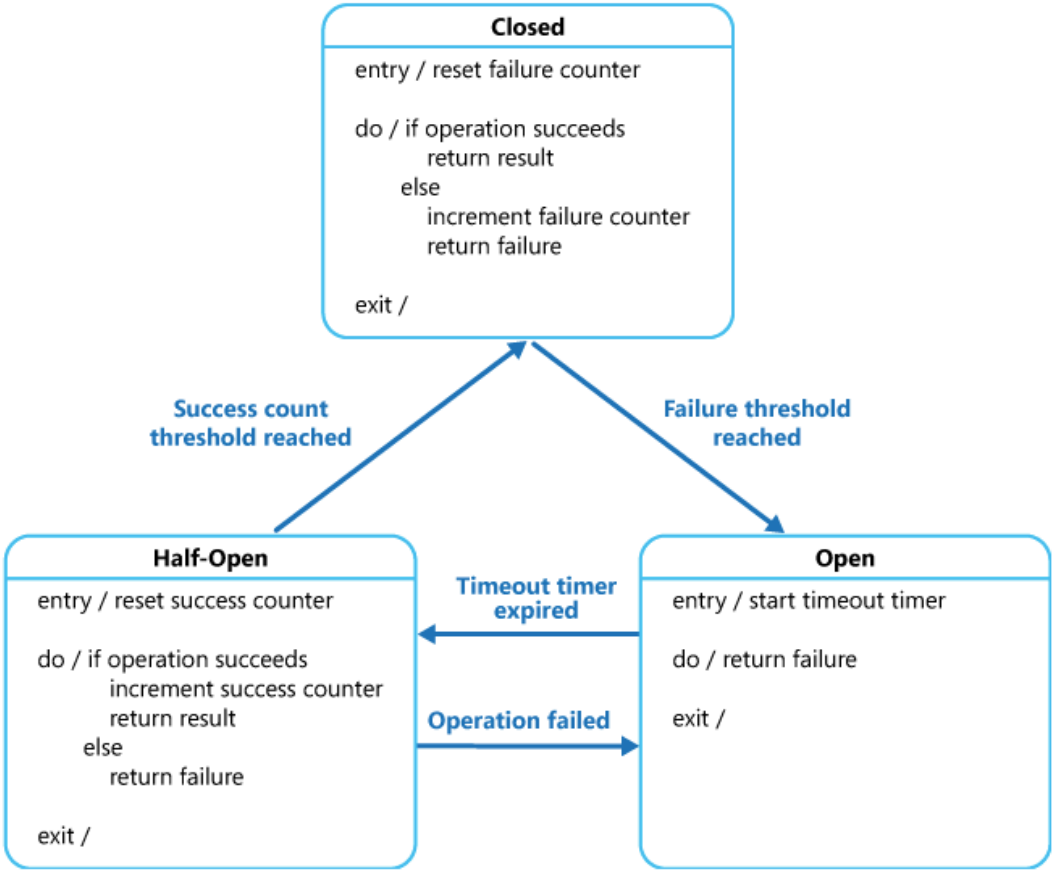
```
@HystrixCommand(fallbackMethod = "fallBackOrderItems")
public List<OrderItem> getOrderItems(String customerId,String orderId) {
        List<OrderItem> orderItems = orderItemRepository.findByOrderId(orderId);
        for(OrderItem i: orderItems) {
                Product prod = productFeignClient.getProduct(i.getProductNumber());
                i.setProductName(prod.getName());
        }
        return orderItems;
}

private List<OrderItem> fallBackOrderItems(String customerId,String orderId) {
        List<OrderItem> orderItems = orderItemRepository.findByOrderId(orderId);
        for(OrderItem i: orderItems) {
                i.setProductName("Generic");}
        return orderItems;
}
```

Hystrix Defaults

- Timeout for every request to an external system (default: 1000 ms)
- Limit of concurrent requests for external system (default: 10)
- Circuit breaker to avoid further requests (default: when more than 50% of all requests fail)
- Retry of a single request after circuit breaker has triggered (default: every 5 seconds)
- Interfaces to retrieve runtime information on request and aggregate level (there's even a ready-to-use realtime dashboard for it)

Breaker States

CP



Handshaking pattern

The Circuit Breaker pattern is an all-or-nothing approach.

- Depending on the quality and granularity of the recorded metrics an alternative is to detect an overload situation in advance. If an impending overload is detected, the client can be signaled to reduce requests.
- In the Handshaking pattern a server communicates with the client in order to control its own workload.
- We have seen an approach to Handshaking in this stack. Servers provide regular system health updates to the Registry, and then to client. Only healthy servers are addressed by client. Other servers may recover as their load diminishes.

BulkHead – when you can't avoid Icebergs

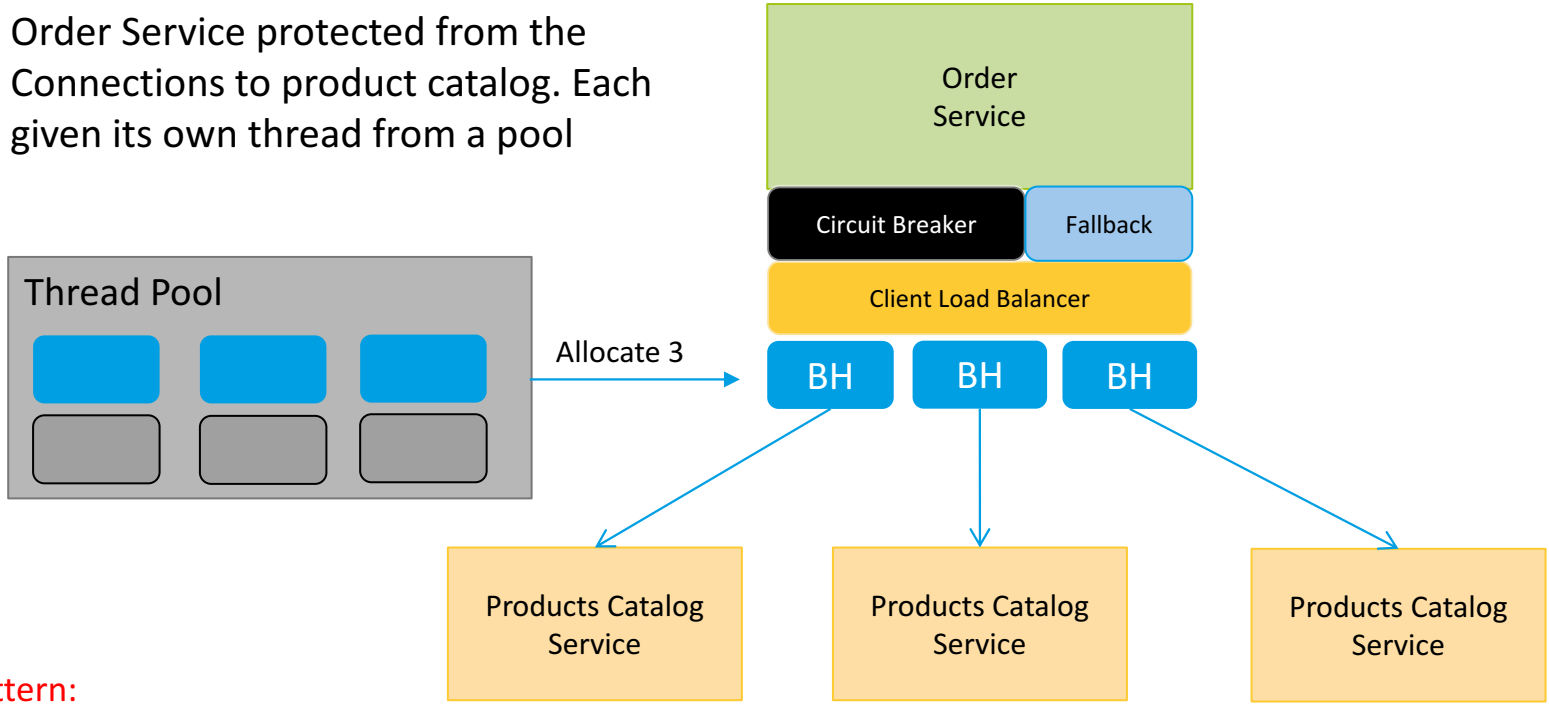
Analogy, a ship is built based on completely segregated water tight compartments, except for the Titanic.

Apply the same to a service that has to interact with multiple remote resources.

- Break the calls to remote resources into their own thread pools.
- Reduce the risk that a problem with one slow remote resource call will take down the entire application.
- Each remote resource is segregated and assigned to the thread pool.
- The thread pools act as the bulkheads for your service.

Building Thread Management is massively complex, Spring Cloud and Hysterix provides this for free, with simple annotations.

Order Service protected from the Connections to product catalog. Each given its own thread from a pool

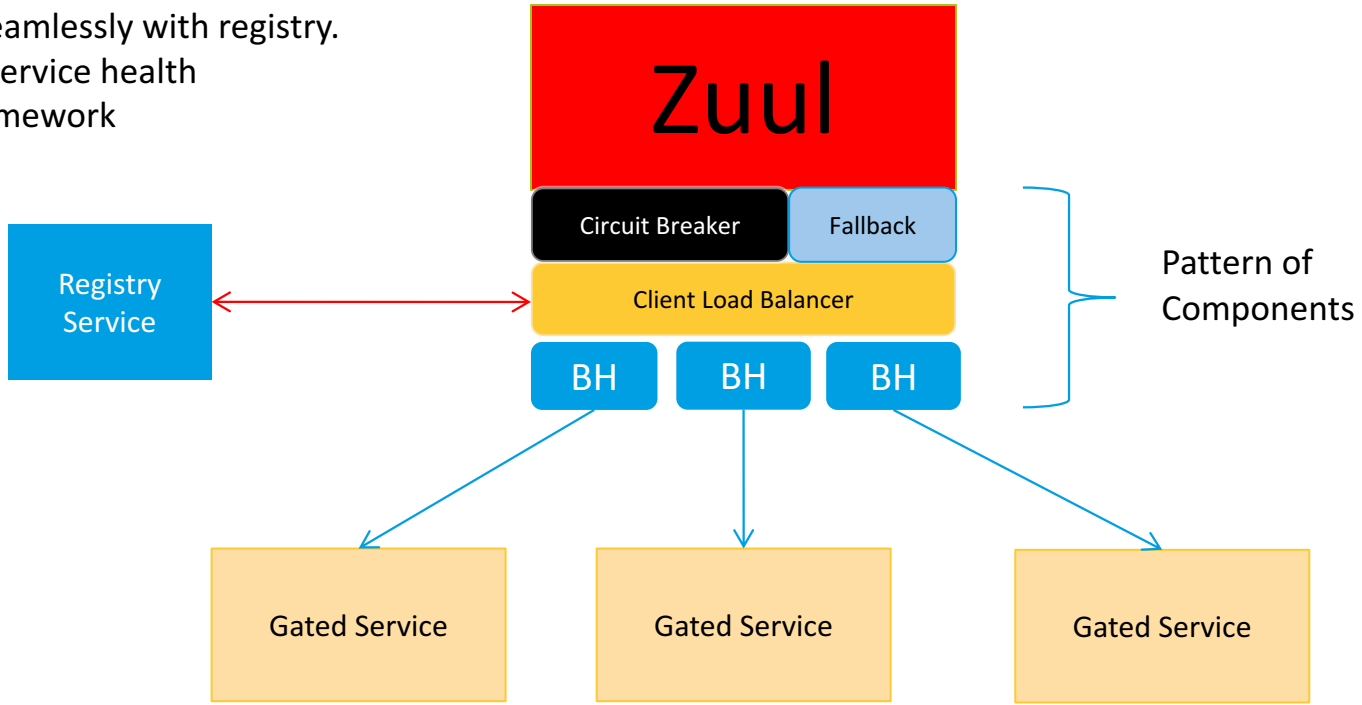


Pattern:

- BulkHead

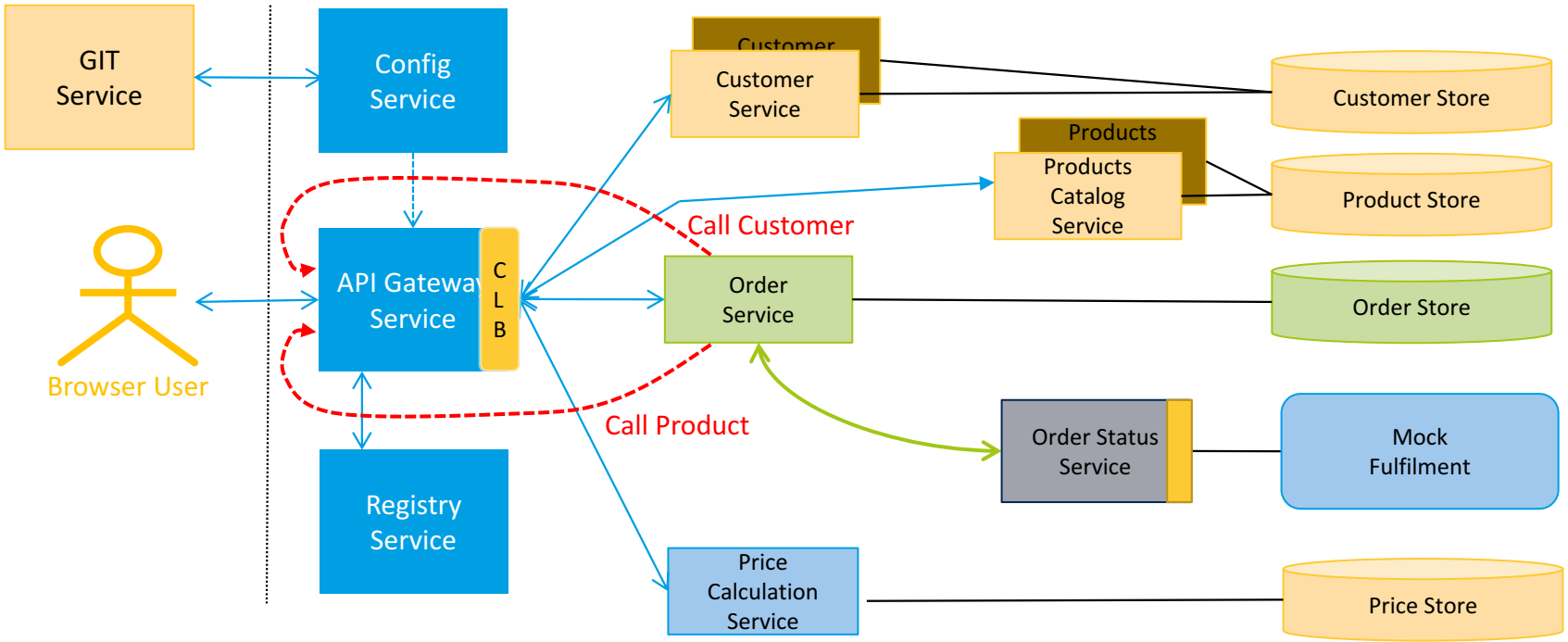
Zuul Revisited – A Cohesive Consistent Framework

- Zuul uses the same pattern of 'protective' and scaling components as the services.
- It integrates seamlessly with registry.
- Knows about service health
- A cohesive framework



Pattern: Gateway as Load Balancing

RP



Pattern: Gateway as Load Balancer – all calls go via Gateway

Hystrix Easy As…..

One annotation protects this method with a Circuit Breaker, Fall Back, and wraps a Bulkhead

```
@HystrixCommand
public List<Cart> getOrdersByCustomer(String customerId){
   return orderRepository.findByCustomerId( customerId );
}
```

Also adds the ‘/health’ endpoint and provides a stream for monitoring.

Monitoring

Hystrix Health and Stream Data

Health Indicator

The state of the connected circuit breakers are also exposed in the /health endpoint of the calling application.

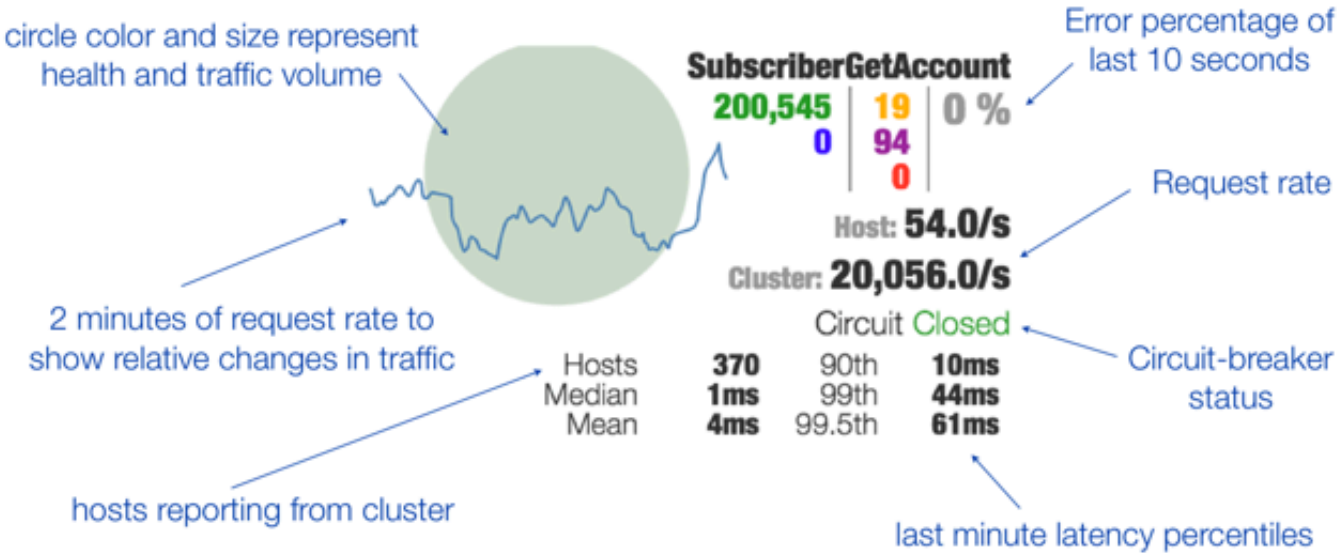
```
{ "hystrix":
    { "openCircuitBreakers": [
    "StoreIntegration::getStoresByLocationLink" ],
    "status": "CIRCUIT_OPEN" },
"status": "UP" }
```

Hystrix Metrics Stream Data

To enable the Hystrix metrics stream include a dependency on spring-boot-starter-actuator.

This will expose the /hystrix.stream as a management endpoint. Data is now streamed about circuit status.

Hystrix Dashboard – Visualise Stream



Rolling 10 second counters
with 1 second granularity

Successes	200,545	19	Thread timeouts
Short-circuited (rejected)	0	94	Thread-pool Rejections
		0	Failures/Exceptions

Monitor Order Service – All OK

Hystrix Stream: http://localhost:8080/hystrix.stream

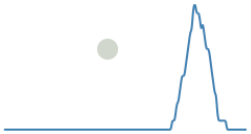
HYSTRIX
DEFEND YOUR APP

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

Success | Short-Circuited | Timeout | Rejected | Failure | Error %

getOrdersByCustomer

0 0 0 0 0 0.0 %

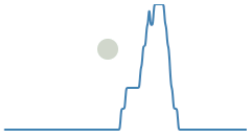


Host: 0.0/s
Cluster: 0.0/s
Circuit Closed

Hosts	1	90th	57ms
Median	32ms	99th	75ms
Mean	37ms	99.5th	75ms

getOrder

0 0 0 0 0 0.0 %



Host: 0.0/s
Cluster: 0.0/s
Circuit Closed

Hosts	1	90th	88ms
Median	59ms	99th	160ms
Mean	68ms	99.5th	160ms

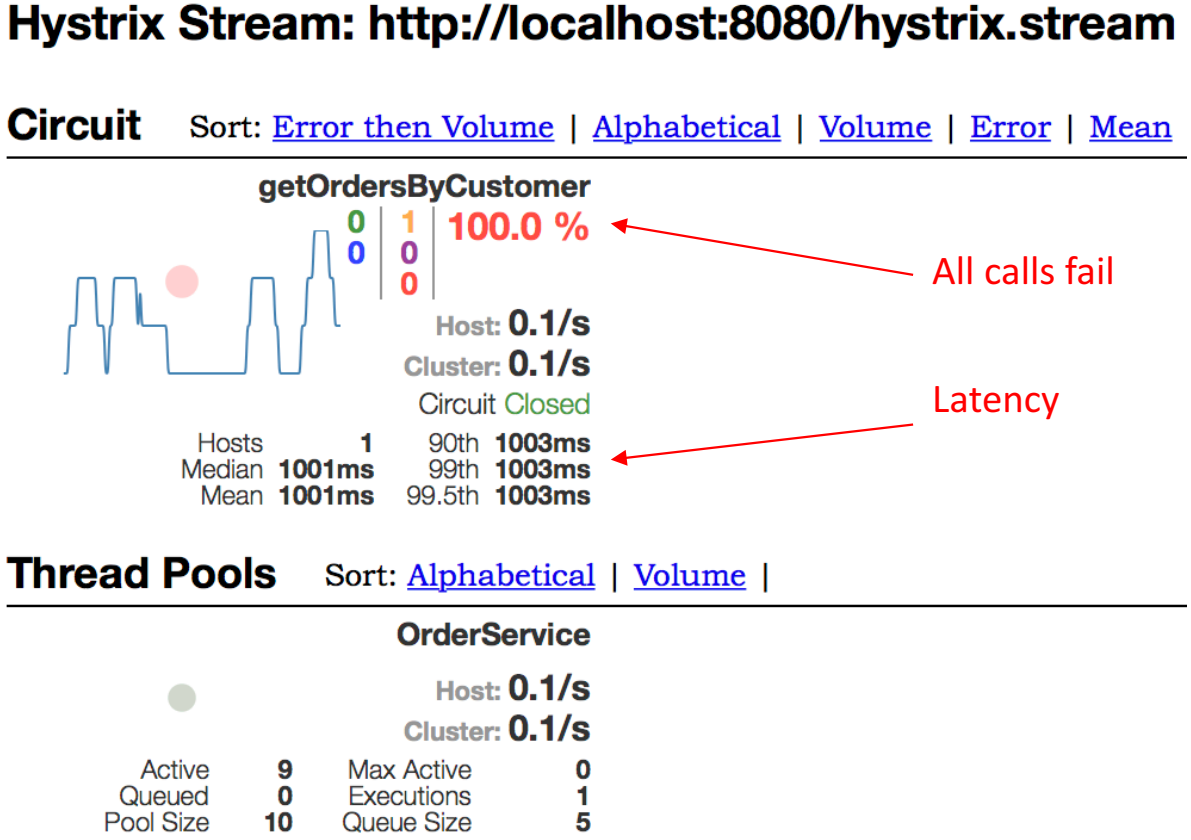
Thread Pools Sort: Alphabetical | Volume |

OrderService

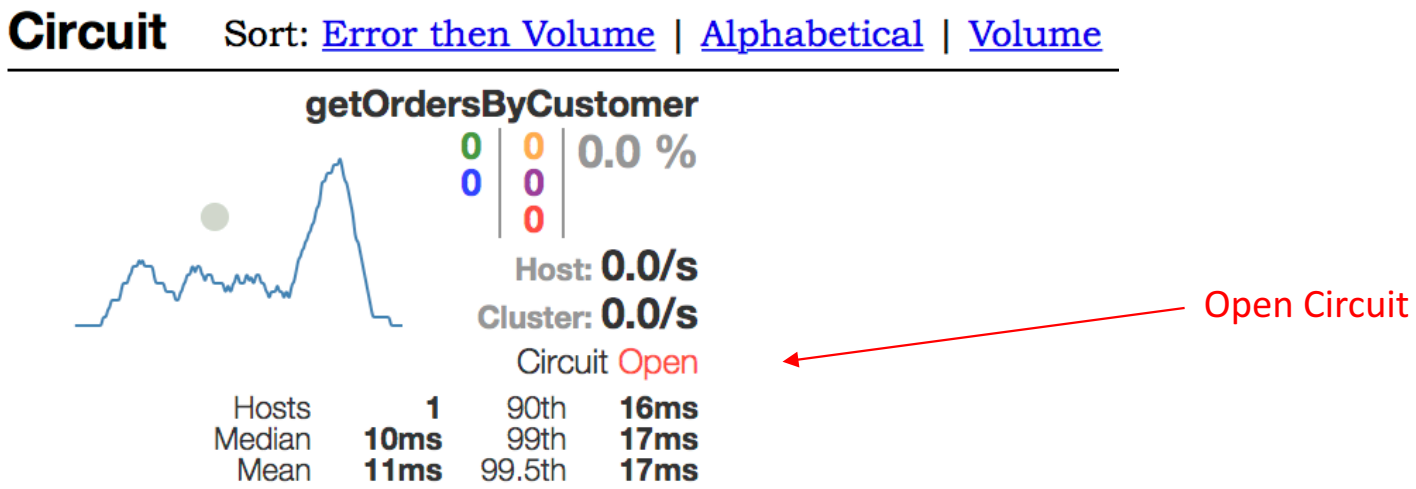
Host: 0.0/s
Cluster: 0.0/s

Active	0	Max Active	0
Queued	0	Executions	0
Pool Size	10	Queue Size	5

Customer Service Stopped

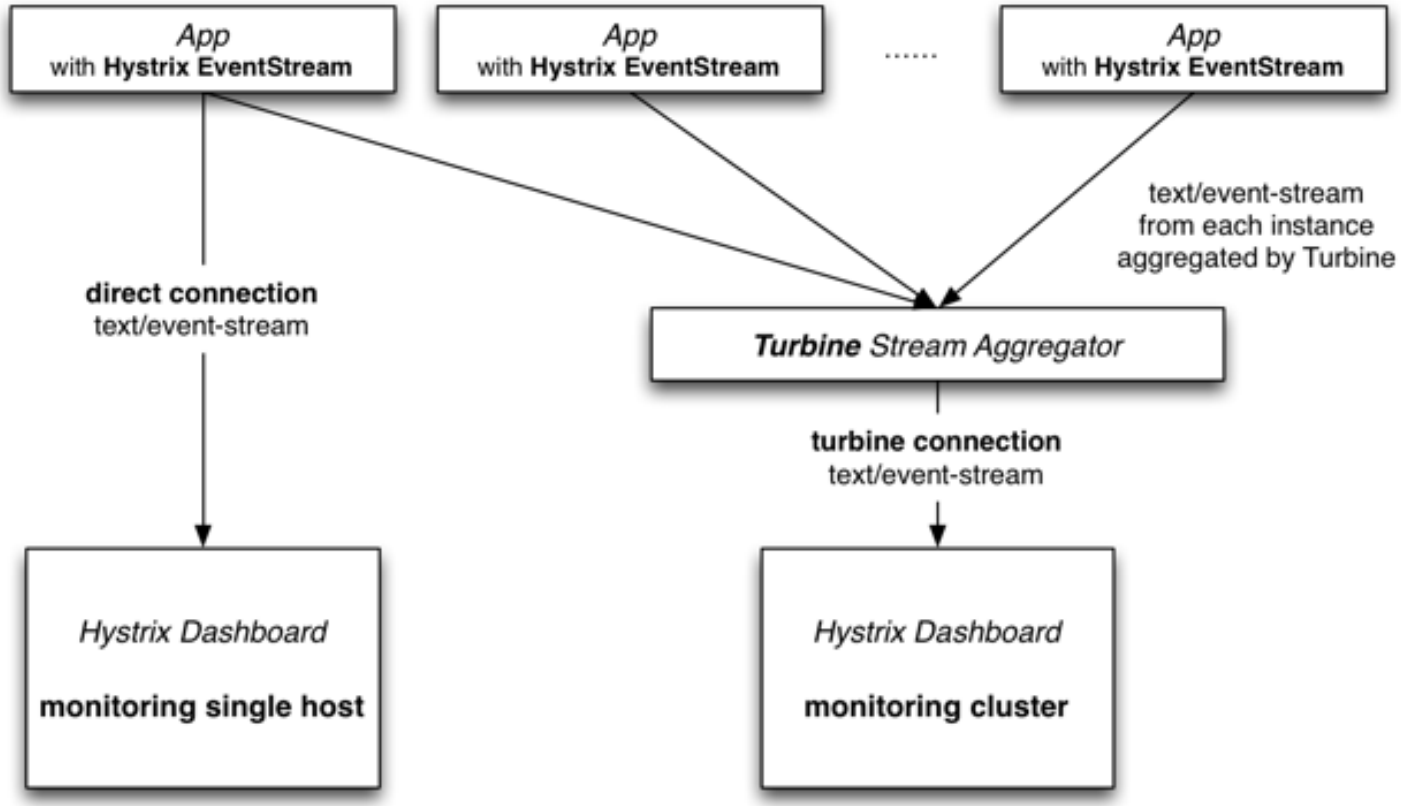


Keep Trying goes Open Circuit



Open Circuit

Turbine Stream Aggregator



Turbine – Aggregate Hystrix Streams

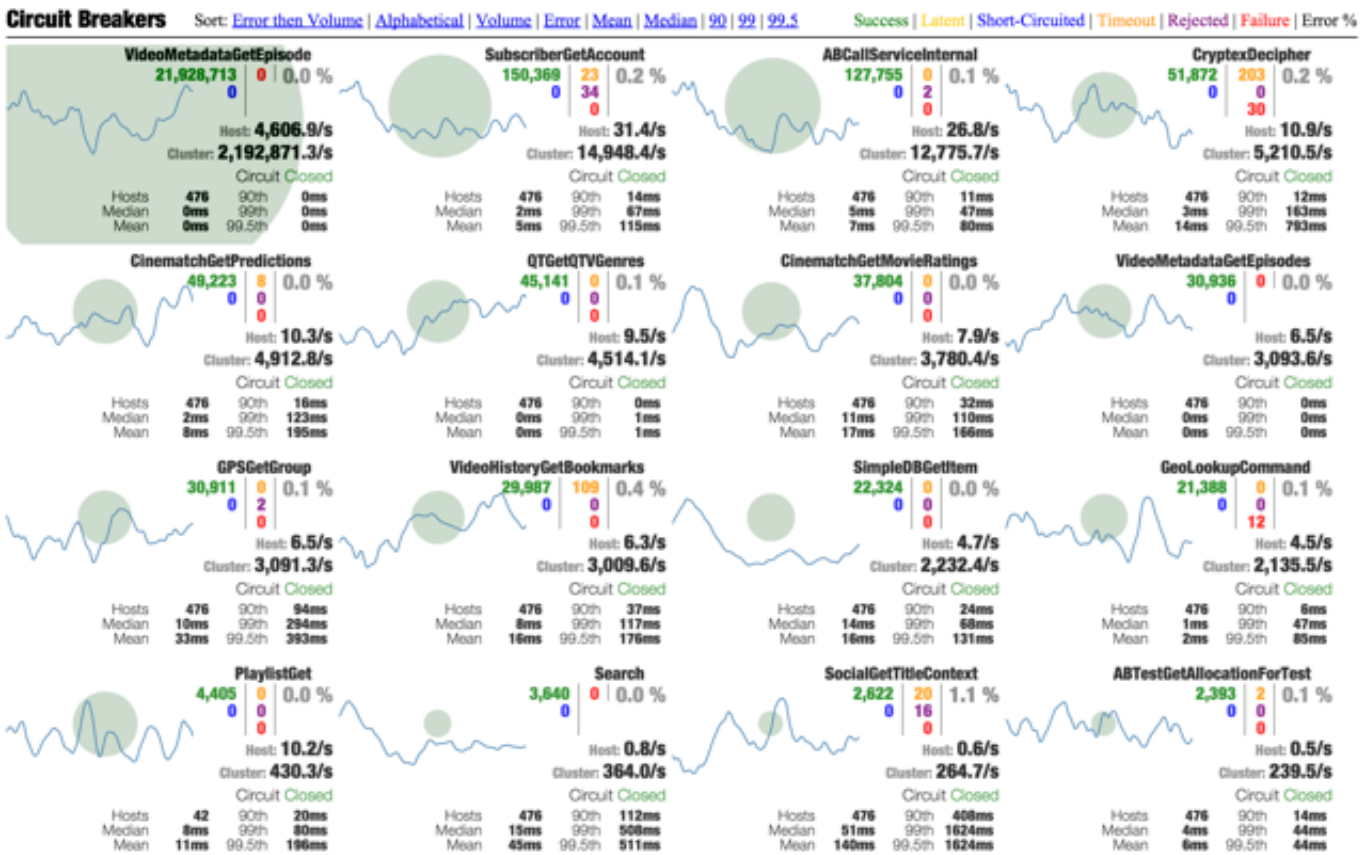
Every hystrix enabled application produces a stream where the status of all circuits are constantly written.

- The application stream contains circuit information of the given application, the URL is:
  http://application-host/hystrix.stream

- Once turbine is running it will produce it's stream at
  http://turine-host/turbine.stream

- The stream can be monitored from the dashboard at:
  http://dashboard-host/hystrix/monitor?stream=http://turbine-host/turbine.stream

Turbine Dashboard



Distributed Tracing / Logging

Sleuth and Zipkin

"Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud. All your interactions with external systems should be instrumented automatically. You can capture data simply in logs, or by sending it to a remote collector service.", such as Zipkin.

Sleuth is based on the MDC (Mapped Diagnostic Context) concept, where you can easily extract values put into a new context (where values are now not in scope) and display them in the logs.

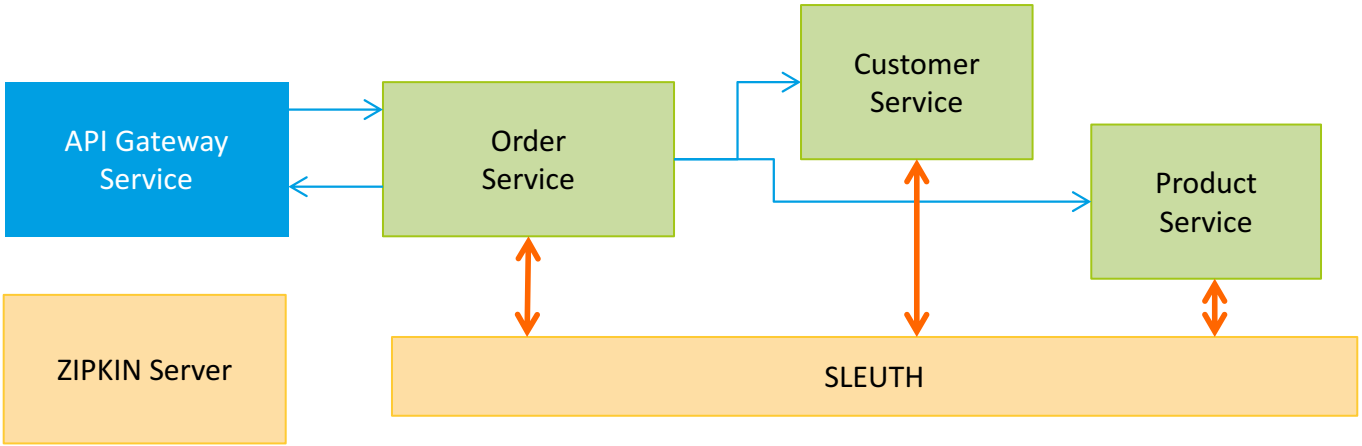
Adding values into a different context improves debugging and tracing significantly.

Zipkin is "*a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data through a Collector and a Query service.*"

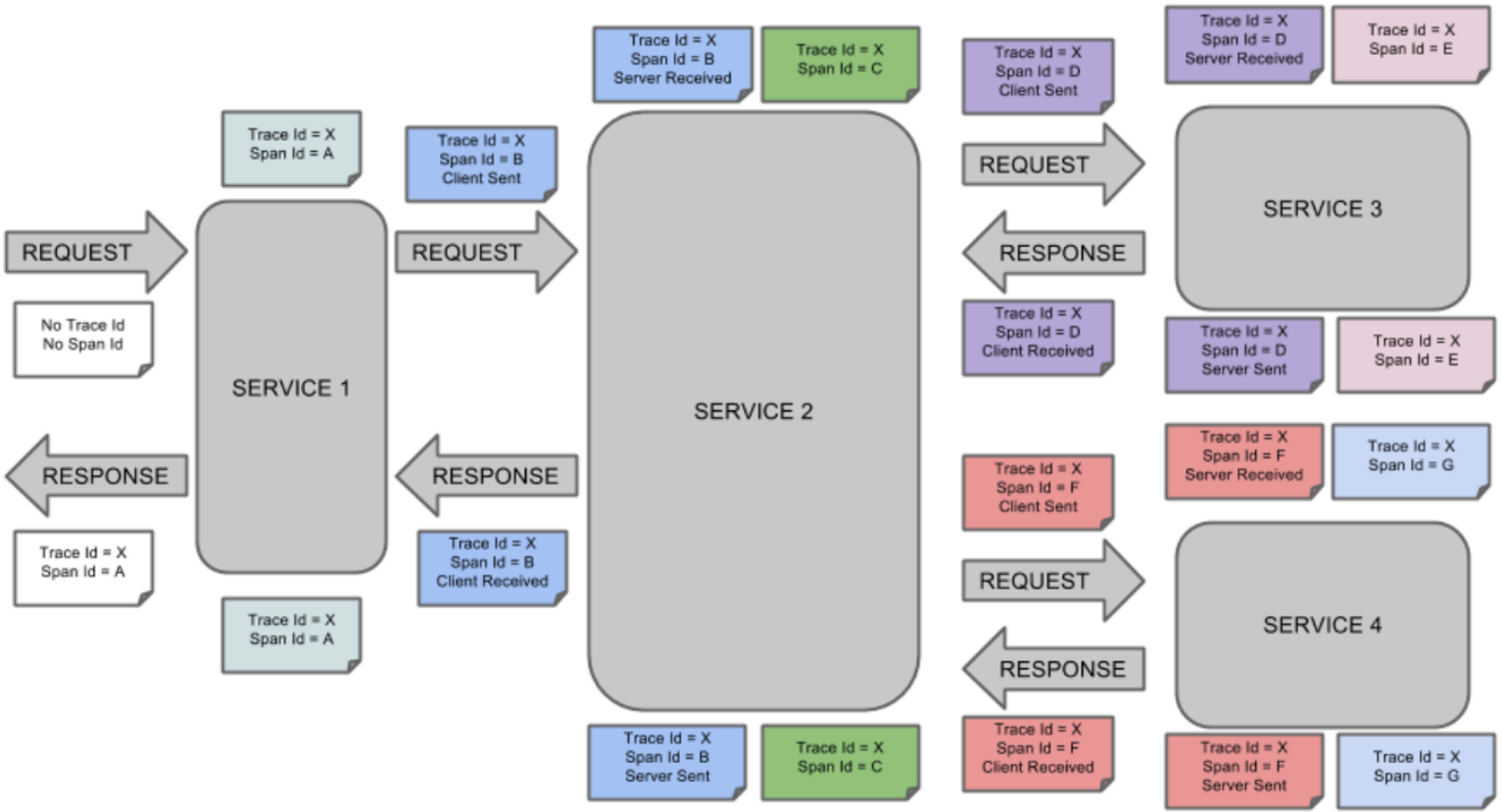
Zipkin provides critical insights into how microservices perform in a distributed system.

Adding Tracing IDS

- Spring Cloud Sleuth adds in Span and Trace IDs:
  - A Span is the basic unit of work, that is timed (Request-Response);
  - A Trace is a set of spans forming a tree-like structure across service calls.
- Only requires adding dependency in POM, no coding !
- ZIPKIN visually correlates the Trace IDs.

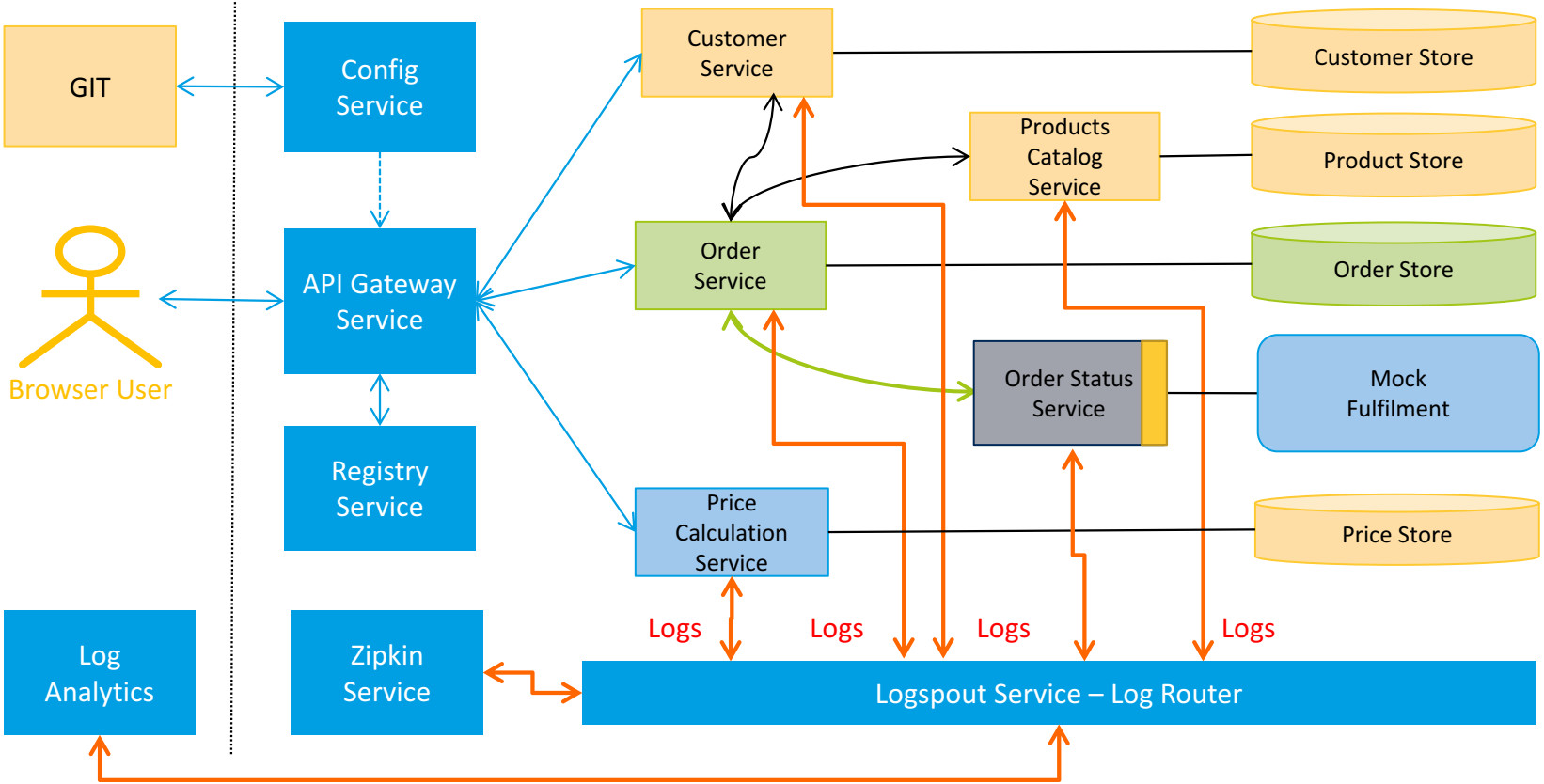


Sleuth – Spans and Trace IDs

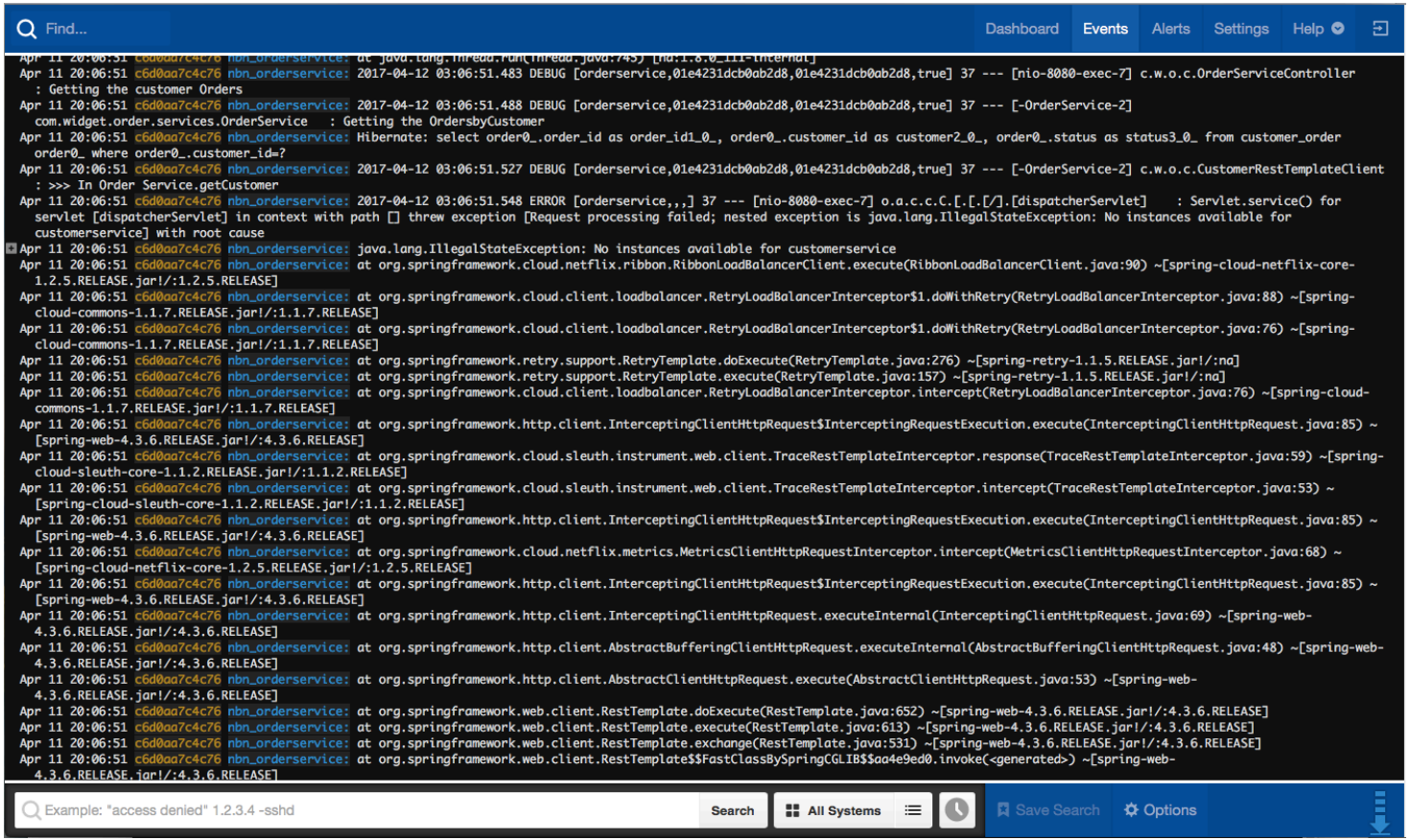


Logging + Tracing - 13 Services

RP



Papertrail via Logspout – Log Analysis

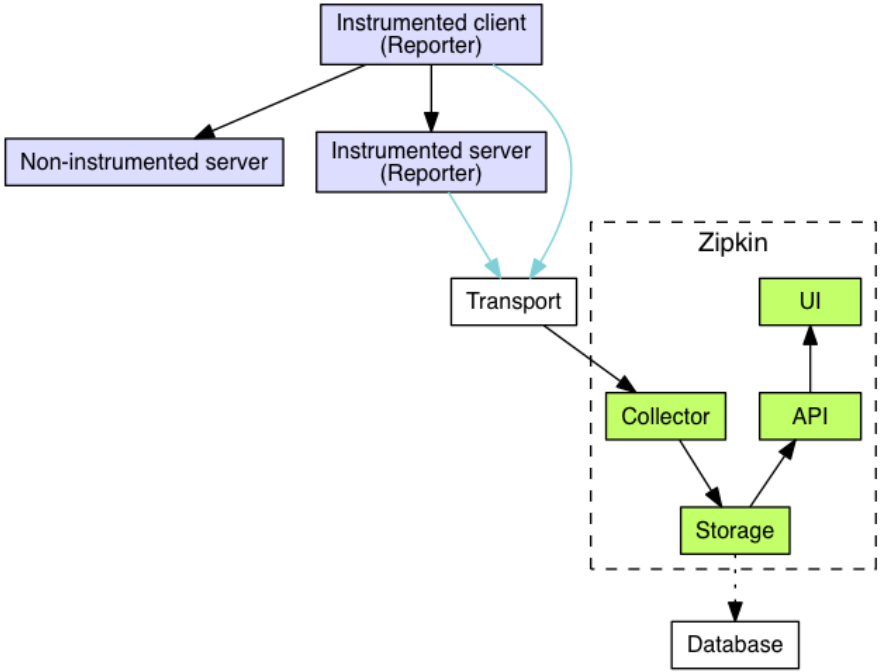


Open Zipkin Project

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems microservice architectures.

Open Tracing Project.

- is an effort to standardize the vocabulary and concepts of modern tracing for multiple languages and platforms.
- Consistent, expressive, vendor-neutral APIs for distributed tracing and context propagation.



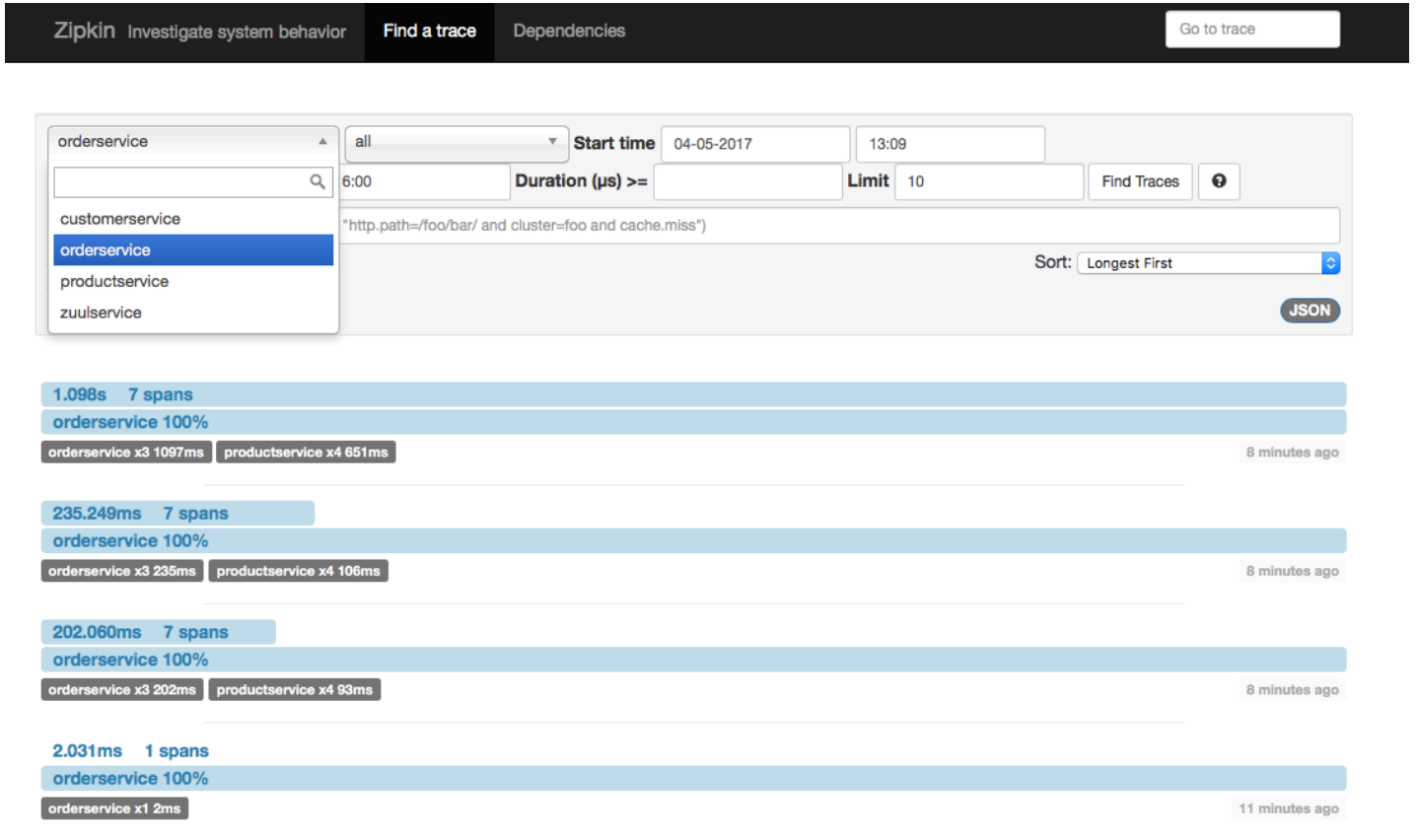
Example Logs with Sleuth Tracing

2016-02-02 15:30:57.902 INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
2016-02-02 15:31:01.936 INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 --- [nio-8081-exec-4] ...

Notice the [appname,traceId,spanId,exportable] entries from the MDC:

- **spanId** - the id of a specific operation that took place
- **appname** - the name of the application that logged the span
- **traceId** - the id of the latency graph that contains the span
- **exportable** - whether the log should be exported to Zipkin or not. When would you like the span not to be exportable? In the case in which you want to wrap some operation in a Span and have it written to the logs only.

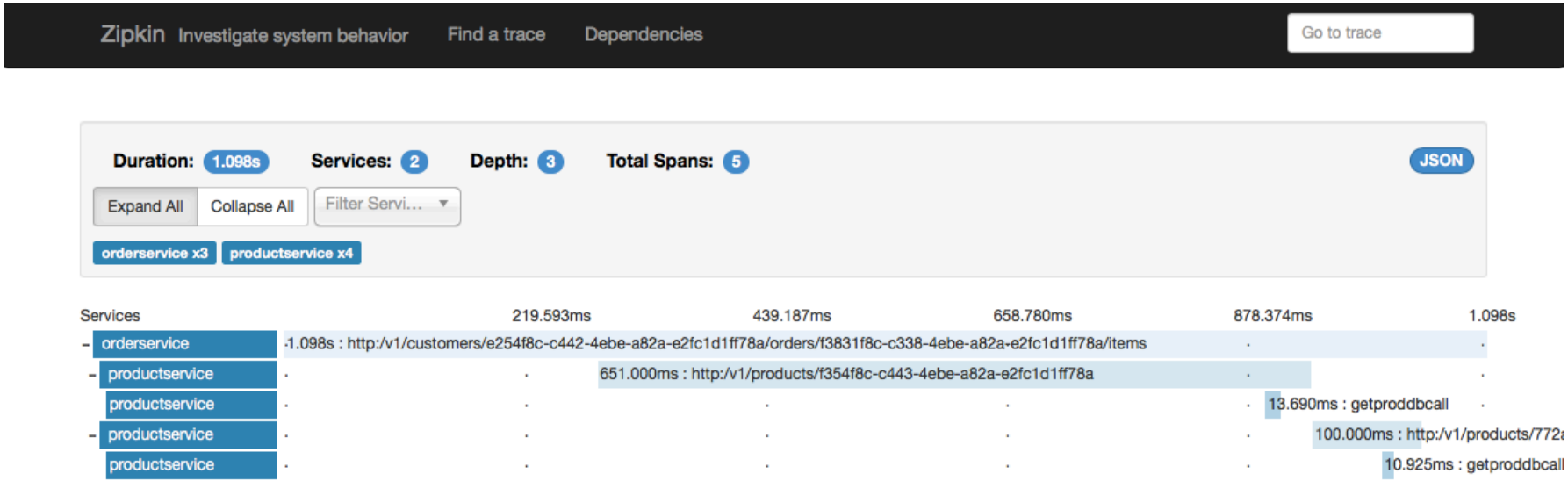
Zipkin: http://localhost:9411



Zipkin Tracing

Getting two Product Items in an Order:

http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a/items



HTTP Header – Add Correlation ID

Use Zuul Post Filter to pass CorrelationID onto External Consumers, if required*.

HTTP Header:

- Content-Type →application/json;charset=UTF-8
- Date →Wed, 19 Apr 2017 04:25:53 GMT
- Transfer-Encoding →chunked
- X-Application-Context →zuulservice:default:5555
- my-correlation-id →de4971c9358bc781

* May be a security concern for some organisations

Zuul - Pass on Correlation ID

my_zuulserver | 2017-04-19 04:25:49.234 INFO [zuulservice,de4971c9358bc781,de4971c9358bc781,true] 43 --- [nio-5555-exec-1] o.s.c.n.zuul.web.ZuulHandlerMapping : Mapped URL path [/configserver/**] onto handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]

my_zuulserver | 2017-04-19 04:25:49.235 INFO [zuulservice,de4971c9358bc781,de4971c9358bc781,true] 43 --- [nio-5555-exec-1] o.s.c.n.zuul.web.ZuulHandlerMapping : Mapped URL path [/customerservice/**] onto handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]

my_zuulserver | 2017-04-19 04:25:49.431 INFO [zuulservice,de4971c9358bc781,f5ad7be7d8e982b0,true] 43 --- [nio-5555-exec-1] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@4a619307: startup date [Wed Apr 19 04:25:49 GMT 2017]; parent: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@5afa04c

my_zuulserver | 2017-04-19 04:25:49.936 INFO [zuulservice,de4971c9358bc781,f5ad7be7d8e982b0,true] 43 --- [nio-5555-exec-1] f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject' annotation found and supported for autowiring

my_customerservice | 2017-04-19 04:25:52.132 INFO [customerservice,,,] 44 --- [nio-8085-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'

_customerservice | 2017-04-19 04:25:52.516 DEBUG [customerservice,de4971c9358bc781,f5ad7be7d8e982b0,true] 44 --- [nio-8085-exec-1] c.w.c.c.CustomerServiceController : Entering the customer-service-controller

my_customerservice | 2017-04-19 04:25:52.519 DEBUG [customerservice,de4971c9358bc781,57f3e2774e13c1b0,true] 44 --- [nio-8085-exec-1] c.w.customer.services.CustomerService : In the CustomerService.getCust call

my_customerservice | 2017-04-19 04:25:52.613 INFO [customerservice,de4971c9358bc781,57f3e2774e13c1b0,true] 44 --- [nio-8085-exec-1] o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory

my_customerservice | Hibernate: select customer0_.customer_id as customer1_0_, customer0_.contact_email as contact_2_0_, customer0_.contact_name as contact_3_0_, customer0_.contact_phone as contact_4_0_, customer0_.name as name5_0_ from customer customer0_ where customer0_.customer_id=?

my_zuulserver | 2017-04-19 04:25:53.741 DEBUG [zuulservice,de4971c9358bc781,de4971c9358bc781,true] 43 --- [nio-5555-exec-1] c.widget.zuulsvr.filters.ResponseFilter : Adding the correlation SPAN id to the outbound headers: de4971c9358bc781

my_zuulserver | 2017-04-19 04:25:53.746 DEBUG [zuulservice,de4971c9358bc781,de4971c9358bc781,true] 43 --- [nio-5555-exec-1] c.widget.zuulsvr.filters.ResponseFilter : Completing outgoing request for: /customerservice/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a.

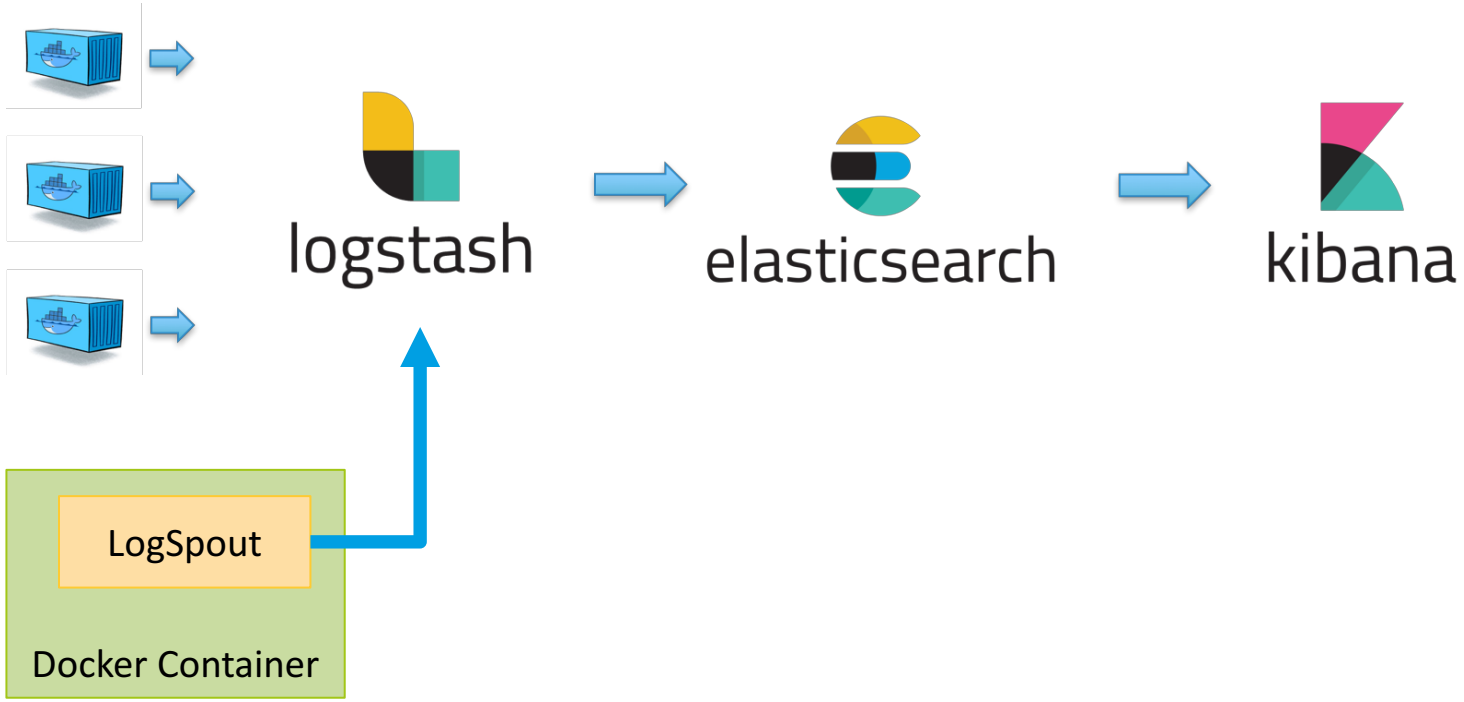
LogSpout

Logspout is a popular and lightweight (15.2MB) log router for Docker that will attach itself to all the containers within a host and output the streams of Docker logs to a syslog server (unless you define a different output destination).

You can use a Logstash module to route directly to your Logstash container, but this requires additional configurations and compilation work on your part.

```
1 sudo docker run -d --name="logspout" --
volume=/var/run/docker sock:/var/run/docker sock gliderlabs/logspout syslog+tls://:5000
```

ELK via Logspout



Log Aggregation Products

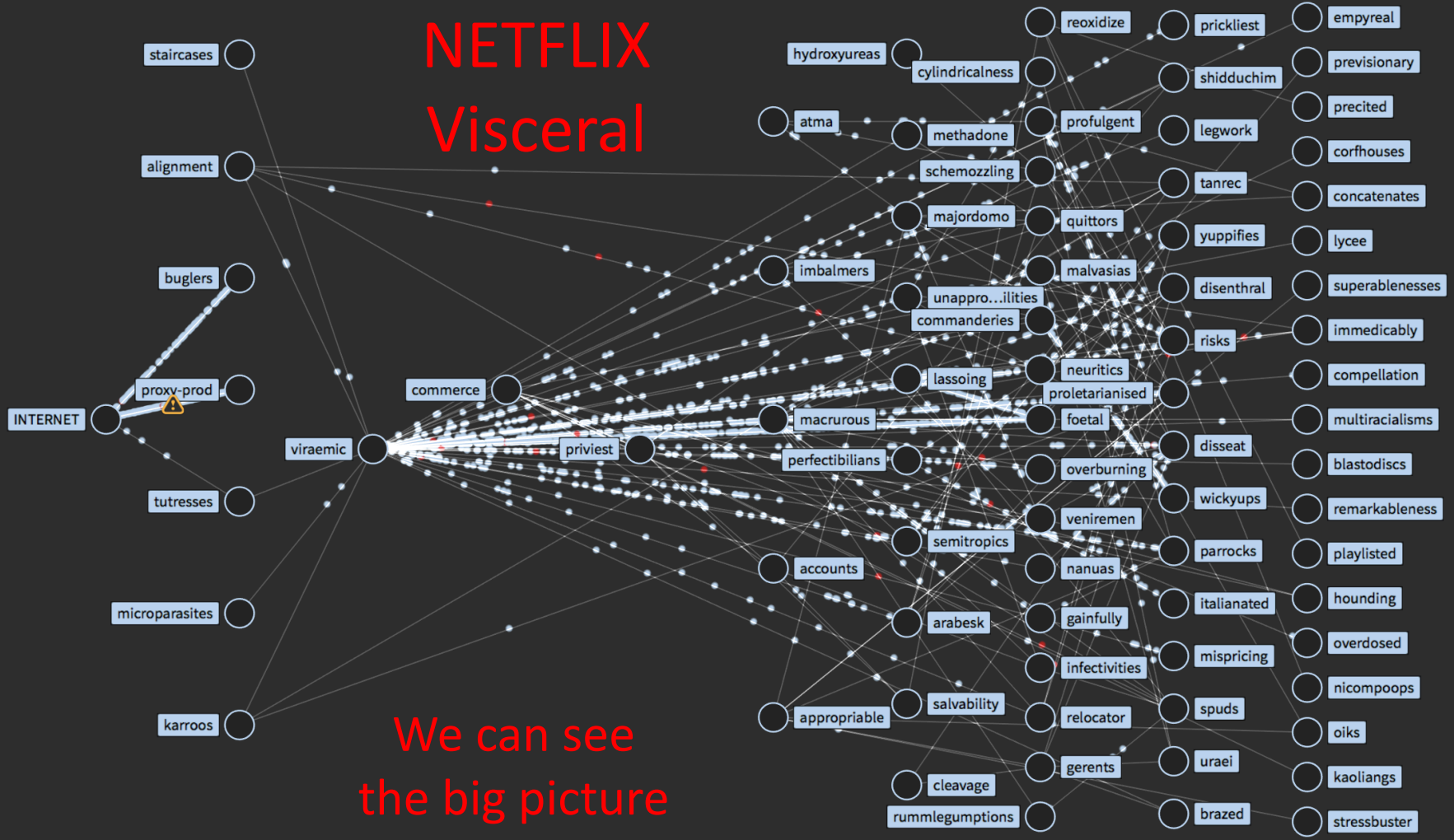
Product	Approach	Description
Elasticsearch, Logstash, Kibana (ELK)	Open source Commercial Typically implemented on premise	• http://elastic.co • General purpose search engine • Can do log-aggregation through the (ELK-stack) • Requires the most hands-on support
Graylog	Open source Commercial On-premise	• http://graylog.org • Open-source platform that's designed to be installed on premise
Splunk	Commercial only On-premise and cloud-based	• http://splunk.com • Oldest and most comprehensive of the log management and aggregation tools • Originally an on-premise solution, but have since offered a cloud offering
Sumo Logic	Freemium Commercial Cloud-based	• http://sumologic.com • Freemium/tiered pricing model • Runs only as a cloud service • Requires a corporate work account to signup
Papertrail	Freemium Commercial Cloud-based	• http://papertrailapp.com • Freemium/tiered pricing model Runs only as a cloud service • Very easy anf fast to get working
Cloudwatch	AWS Only	• AWS Proprietary

Registry Big Picture

Registry and the Big Picture

- Having as centralized registry provide a single knowledge source for:
  - Health information
  - Providers across whole application not localised
  - DNS infrastructure independence

- Provides a way to see the Big Picture view of our application


NETFLIX
Visceral
staircases
alignment
buglers
proxy-prod
INTERNET
tutresses
microparasites
karroos
viraemic
commerce
priviest
hydroxyureas
cylindricalness
atma
methadone
schemozzling
majordomo
imbalmers
unappro...ilities
commanderies
lassoing
macrurous
perfectibilians
semitropics
accounts
arabesk
salvability
appropriable
cleavage
rummlegumptions
reoxidize
profulgent
quittors
malvasias
neuritics
proletarianised
foetal
overburning
veniremen
nanuas
gainfully
infectivities
relocator
gerents
prickliest
shidduchim
legwork
tanrec
yuppifies
disenthral
risks
disseat
wickyups
parrocks
italianated
mispricing
spuds
uraei
brazed
empyreal
previsionary
precited
corfhouses
concatenates
lycee
superablenesses
immedicably
compellation
multiracialisms
blastodiscs
remarkableness
playlisted
hounding
overdosed
nicompoops
oiks
kaoliangs
stressbuster
We can see
the big picture

The End

OSI Layers and Load Balancing

Layer 7 load balancers operate at the highest level in the OSI model, the *application* layer (on the Internet, HTTP is the dominant protocol at this layer). Routing decisions are based on various characteristics of the HTTP header and on the actual contents of the message, such as the URL, the type of data (text, video, graphics), or information in a cookie.

- **Internet Protocol (IP)** operates at the *internetwork* layer (Layer 3). Its PDUs are called *packets*, and IP is responsible for delivering them from a origin host to a destination host, usually across the boundaries between the multiple smaller networks that make up the Internet. Each device that is directly connected to the Internet has a unique IP address, which is used to locate the device as the recipient of packets.
- **Transmission Control Protocol (TCP)** operates at the *transport* layer (Layer 4). TCP effectively creates a virtual connection between the host where the browser is running and the host where a server application is running. Because of the unreliable nature of networks, IP packets can be lost, corrupted, or arrive out of order. TCP has mechanisms for correcting these errors, transforming the stream of IP packets into a reliable communication channel. Each application is assigned a unique TCP port number to enable delivery to the correct application on hosts where many applications are running.
- **Hypertext Transfer Protocol (HTTP)** operates at the *application* layer (Layer 7). It defines how data is encoded for communication between web browsers and web servers (or any application that understands HTTP encoding).