Introduction to Spring Cloud

Fault Tolerance

LXFT LISTED NYSE

LUXOFT

Hystrix

Hystrix

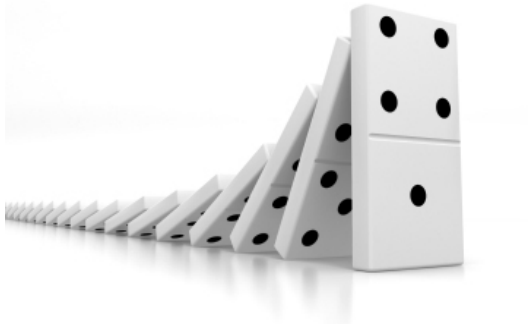
- We have already seen Hystrix circuit breaker at work integrated with Feign REST client.
- However Hystrix is not just a circuit breaker…

Hystrix

- 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;
  - enable resilience in complex distributed systems where failure is inevitable;
- Hystrix, out of the box
  - supports timeouts, load shredding, circuit breaker
  - uses thread pools for dependency services

Hystrix

- Latency & Fault Tolerance
  - Stop cascading failures;
  - Fallbacks and graceful degradation;
  - Fail fast and rapid recovery;
  - Thread and semaphore isolation with circuit breakers.

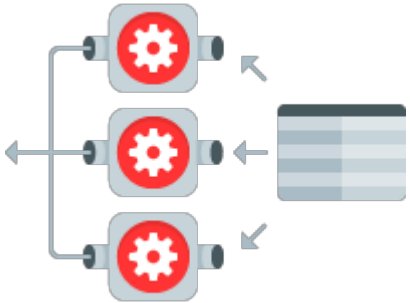


Hystrix

- Real-time operations
  - Real-time monitoring and configuration changes;
  - Watch service and property changes take effect immediately as they spread across a fleet;
  - Be alerted, make decisions, affect change and see results in seconds;

Hystrix

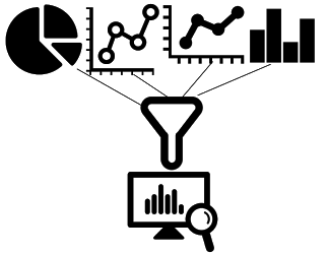
- Concurrency
  - Parallel execution;
  - Concurrency aware request caching;
  - Automated batching through request collapsing;



Hystrix Command

Hystrix Command

- As Hystrix commands execute, they generate metrics on execution outcomes and latency.
  - These are very useful to operators of the system, as they give a great deal of insight into how the system is behaving.
  - Hystrix offers metrics per command key and to very fine granularities (on the order of seconds).

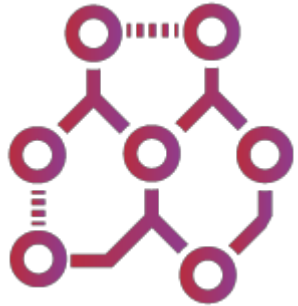


Hystrix Command

- These metrics are useful both individually, and in aggregate.
  - Getting the set of commands that executed in a request, along with outcomes and latency information, is often helpful in debugging.
  - Aggregate metrics are useful in understanding overall system-level behavior, and are appropriate for alerting or reporting.

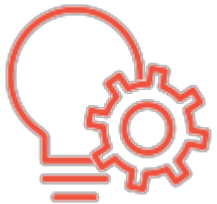
Hystrix Command

- Hystrix commands
  - are used in Feign to guard REST related issues;
  - could be used for any method in code;
- You can use Hystrix commands to wrap
  - a method that may throw an Exception;
  - a method that should be executed within a timeout;



Hystrix Command

- @HystrixCommand annotation represents a single hystrix command.
  - It ships with “spring-cloud-starter-hystrix” dependency.
- Once @HystrixCommand annotation is used, several features of hystrix command are available throw annotation parameters.



Hystrix Command

- @HystrixCommand annotation parameters:
  - fallbackMethod – the method name, result value of which will be used if an Exception is thrown in original method.
    - fallbackMethod accepts a method name from current class (private members included).
  - ignoreExceptions – array of Exception classes to be ignored by hystrix command.

Hystrix Command

- @HystrixCommand annotation parameters (continuation):
  - commandKey – is used for distinction of hystrix commands.
    - hystrix.command.**commandKey**.execution.isolation.thread.timeoutInMilliseconds: 1000
    - *By default the name of command key is command method name.*
  - groupKey – is used to keep commands organized in logical groups.
    - *Default group key name is class name of annotated method.*
  - threadPoolKey – is used to separate command to an exact thread-pool

Hystrix Command

- The thread-pool key represents a hystrix thread-pool for monitoring, metrics publishing, caching, and other such uses.
  - A HystrixCommand is associated with a single thread-pool as retrieved by the threadPoolKey value;
  - or it defaults to one created using the groupKey it is created with;
- The reason why you might use threadPoolKey instead of just a different groupKey is that multiple commands may belong to the same “group” of ownership or logical functionality, but certain commands may need to be isolated from each other.
  - Default thread-pool size is 10

Hystrix Command

- Here is a simple example where threadPoolKey could be used:
  - two commands used to access data storage;
  - group name is “DataStorage”;
  - command A goes against storage #1;
  - command B goes against storage #2;
- If command A becomes latent and saturates its thread-pool it should not prevent command B from executing requests since they each hit different back-end resources.
  - Thus, we logically want these commands grouped together but want them isolated differently and would use threadPoolKey to give each of them a different thread-pool.

Hystrix Command

- A special configuration option “execution.isolation.strategy” of hystrix can take one of two predefined values.
  - THREAD (default)
    - it executes on a separate thread and concurrent requests are limited by the number of threads in the thread-pool
  - SEMAPHORE
    - it executes on the calling thread and concurrent requests are limited by the semaphore count

Hystrix Command

- The default, and the recommended setting, is to run commands using thread isolation (THREAD).
  - Commands executed in threads have an extra layer of protection against latencies beyond what network timeouts can offer.
- Generally the only time you should use semaphore isolation (SEMAPHORE) is when the call is so high volume that the overhead of separate threads is too high
  - hundreds per second, per instance (this typically only applies to non-network calls)
  - SEMAPHORES does not support timeouts!

Hystrix Command

- When a Hystrix command is executed, Hystrix checks with the circuit-breaker to see if the circuit is open.
  - If the circuit is open (or “tripped”) then Hystrix will not execute the command but will execute the Fallback.
  - If the circuit is closed then the flow proceeds to check if there is capacity available to run the command.

Hystrix Command

- If the thread-pool and queue (or semaphore, if not running in a thread) that are associated with the command are full then Hystrix will not execute the command but will immediately execute the Fallback.
  - Netflix API has 100+ commands running in 40+ thread pools and only a handful of those commands are not running in a thread - those that fetch metadata from an in-memory cache or that are façades to thread-isolated commands.

Hystrix Command

- If the method exceeds the command's timeout value, the thread will throw a TimeoutException. In that case Hystrix routes the response through Fallback, and it discards the eventual return value of method if that method does not cancel/interrupt.
- If the command did not throw any exceptions and it returned a response, Hystrix returns this response after it performs some logging and metrics reporting.

Hystrix Command

- Hystrix reports successes, failures, rejections, and timeouts to the circuit breaker, which maintains a rolling set of counters that calculate statistics.
  - It uses these stats to determine when the circuit should “trip,” at which point it short-circuits any subsequent requests until a recovery period elapses, upon which it closes the circuit again after first checking certain health checks.

Hystrix Command

- Hystrix tries to revert to your fallback whenever a command execution fails:
  - when an exception is thrown by method;
  - when the command is short-circuited because the circuit is open;
  - when the command's thread pool and queue or semaphore are at capacity;
  - when the command has exceeded its timeout length.

Hystrix Command

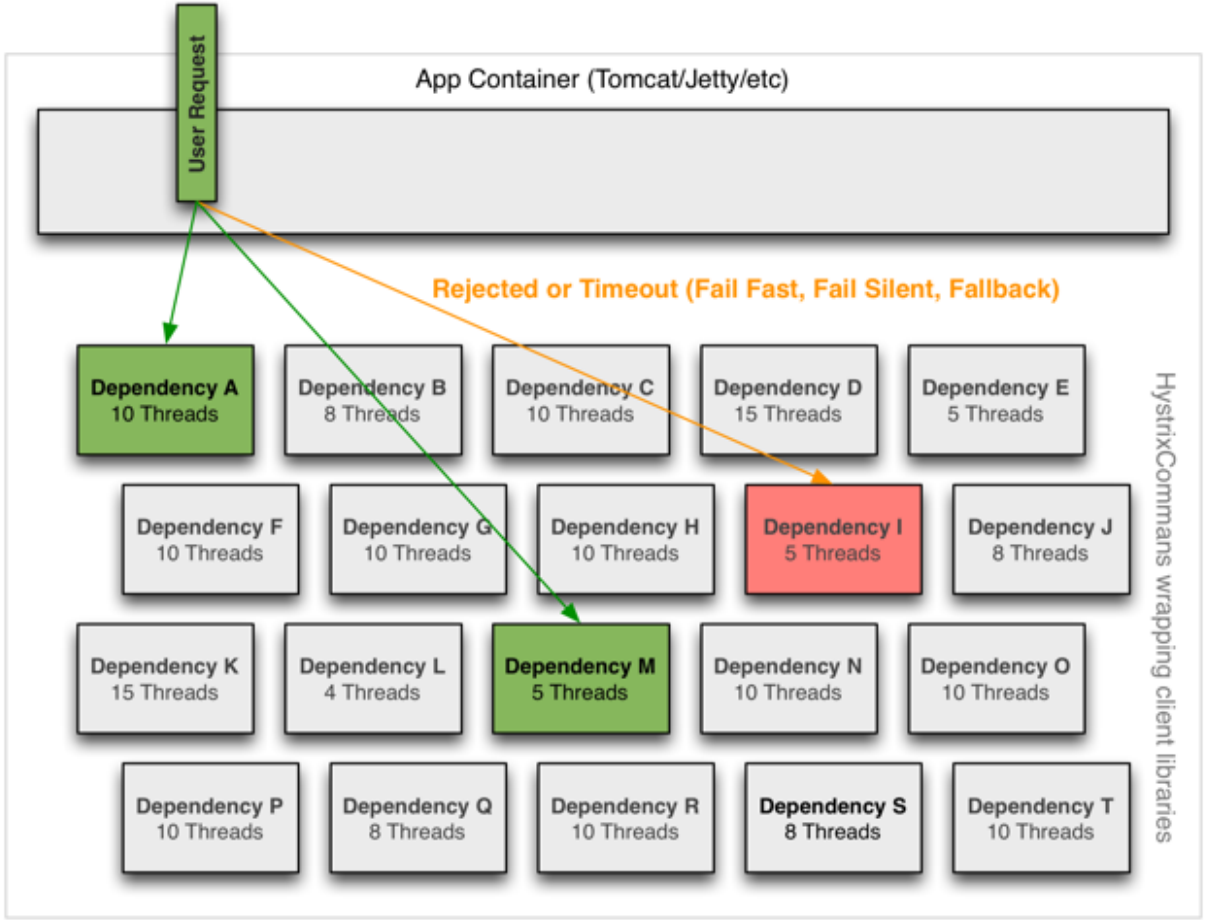
- Write your fallback to provide a generic response, without any network dependency, from an in-memory cache or by means of other static logic.
  - If you must use a network call in the fallback, you should do so by means of another Hystrix command.
- If the fallback method returns a response then Hystrix will return this response to the caller.
  - It is a poor practice to implement a fallback implementation that can fail. You should implement your fallback such that it is not performing any logic that could fail.

Hystrix Command

- The precise way that the circuit opening and closing occurs is as follows:
  1. Assuming the volume across a circuit meets a certain threshold.
  2. And assuming that the error percentage exceeds the threshold error percentage.
  3. Then the circuit-breaker transitions from CLOSED to OPEN.
  4. While it is open, it short-circuits all requests made against that circuit-breaker.
  5. After some amount of time the next single request is let through (this is the HALF-OPEN state).
     - If the request fails, the circuit-breaker returns to the OPEN state for the duration of the sleep window. If the request succeeds, the circuit-breaker transitions to CLOSED and the logic in **1.** takes over again.

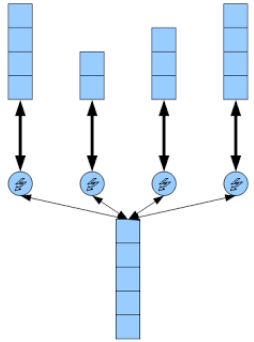
Hystrix Command

- Hystrix employs the bulkhead pattern to isolate dependencies from each other and to limit concurrent access to any one of them.



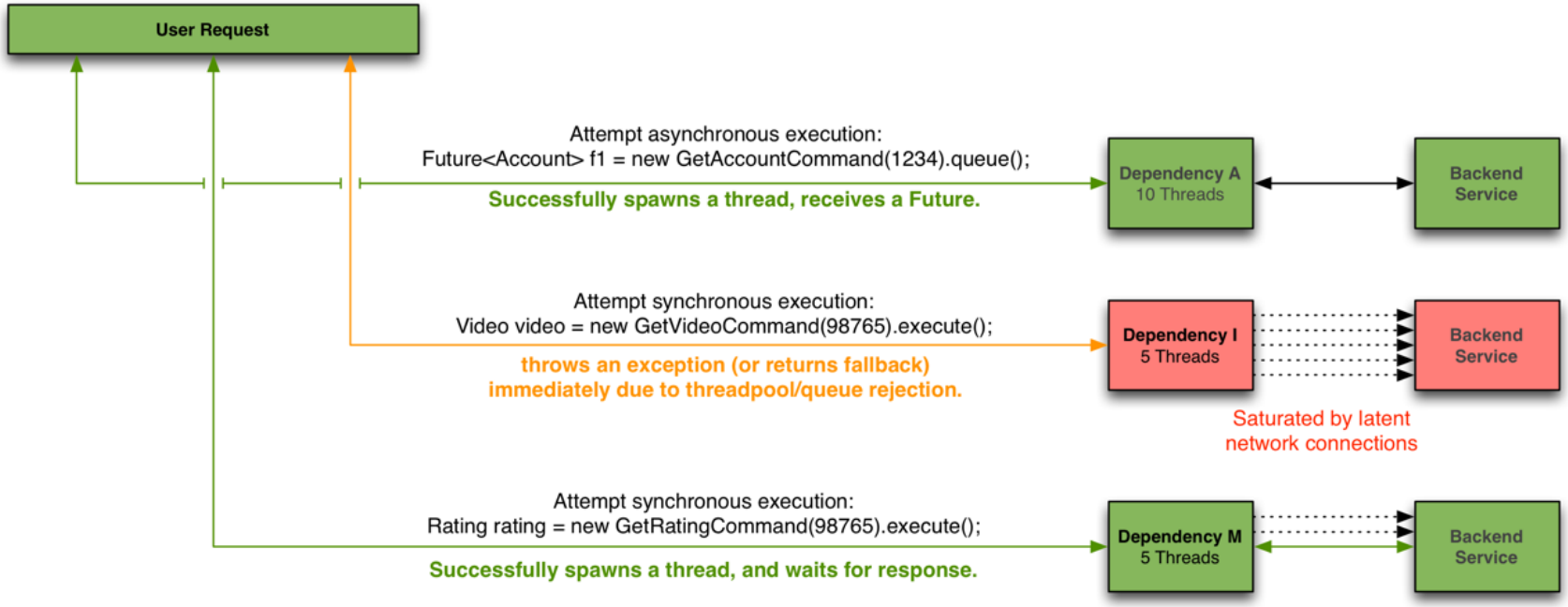
Hystrix Command

- Clients (libraries, network calls, etc) execute on separate threads.
  - This isolates them from the calling thread (Tomcat thread pool) so that the caller may “walk away” from a dependency call that is taking too long.
- Hystrix uses separate, per-dependency thread pools as a way of constraining any given dependency so latency on the underlying executions will saturate the available threads only in that pool.



Hystrix Command

- It is possible for you to protect against failure without the use of thread pools, but this requires the client being trusted to fail very quickly (network connect/read timeouts and retry configuration) and to always behave well.



Hystrix Command

- There are many settings that could be configured for hystrix commands.
  - All options found in official hystrix documentation from Netflix could be applied too spring cloud projects.
- For deep configuration options of thread pools and semaphores see official configuration documentation found at Netflix Hystrix wiki page:
  - https://github.com/Netflix/Hystrix/wiki/Configuration

Hystrix Command

- Some of options available for thread execution with default values:
  - hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 1000
  - hystrix.command.default.execution.timeout.enabled: true
  - hystrix.command.default.execution.isolation.thread.interruptOnTimeout: true
  - hystrix.command.default.execution.isolation.thread.interruptOnCancel: false

Hystrix Command

- To control circuit-breaker options, use:
  - hystrix.command.default.circuitBreaker.enabled: true
  - hystrix.command.default.circuitBreaker.requestVolumeThreshold: 20
  - hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds: 5000
  - hystrix.command.default.circuitBreaker.errorThresholdPercentage: 50
  - hystrix.command.default.circuitBreaker.forceOpen: false
  - hystrix.command.default.circuitBreaker.forceClosed: false

Hystrix Command

- To enable hystrix commands
  - Add hystrix starter dependency to your project

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

  - Add corresponding annotation to your configuration class
    - `@EnableHystrix`

Hystrix Stream

Hystrix Stream

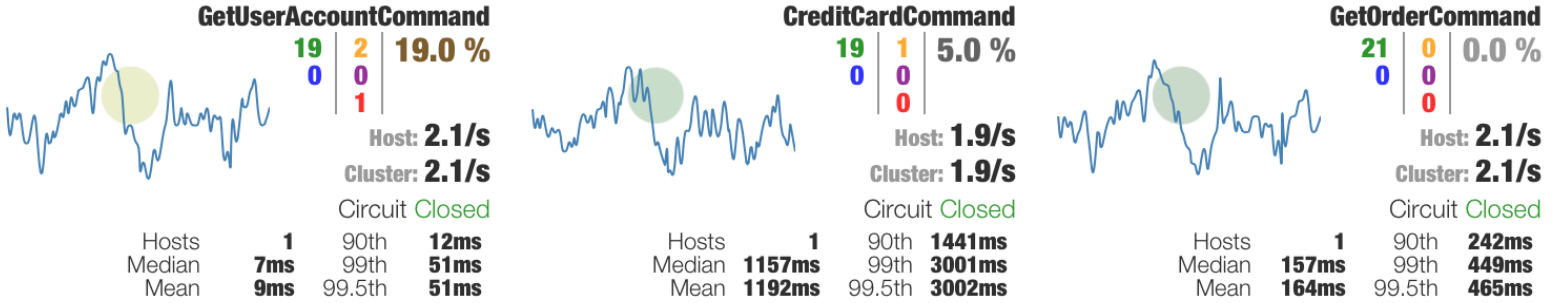
- Hystrix clients are instrumented to produce hystrix streams.
  - Using a hystrix stream one can monitor all commands in the application.
- For SpringBoot applications it is necessary to declare Actuator starter dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- Once actuator and hystrix are on classpath, the hystrix stream will be available at **`/hystrix.stream`** endpoint

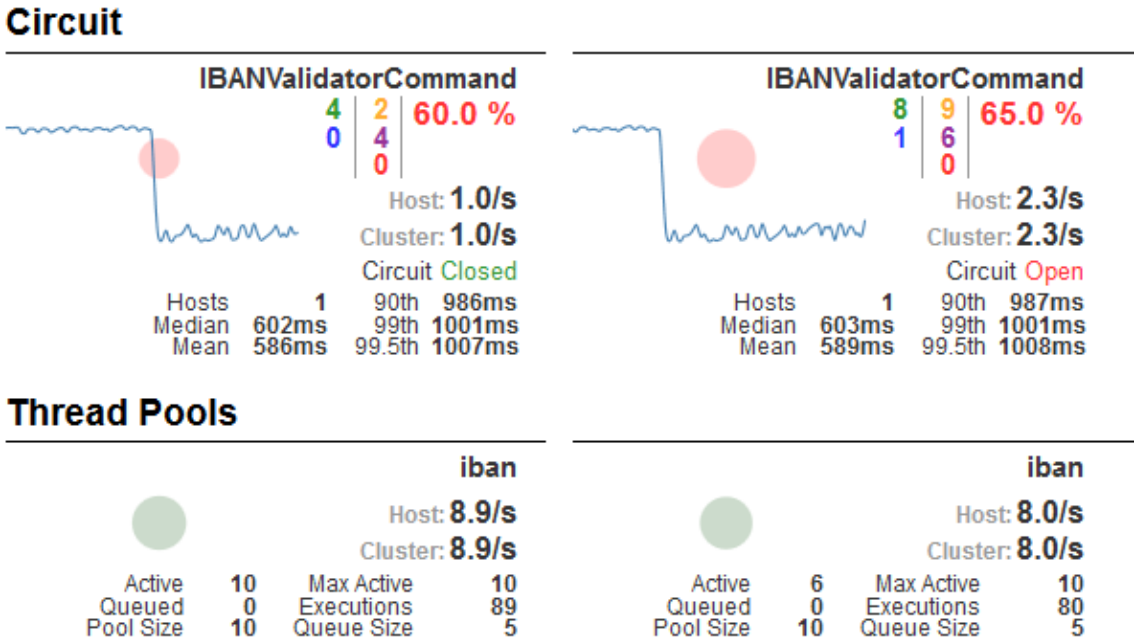
Hystrix Stream

- Hystrix stream provides real-time metrics on the state of the Hystrix commands in an application
- This data tends to be very raw though, a very cool interface called the Hystrix Dashboard consumes this raw data and presents a graphical information based on the underlying raw data

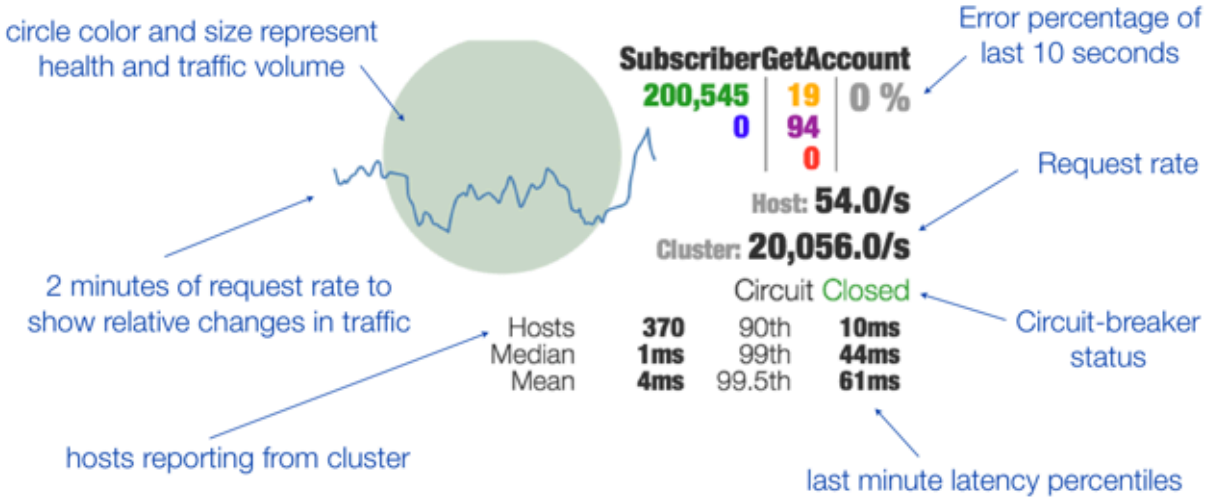


Hystrix Dashboard

- Hystrix dashboard analyzes available hystrix stream data and provides real-time visualization of hystrix commands' status.



Hystrix Dashboard

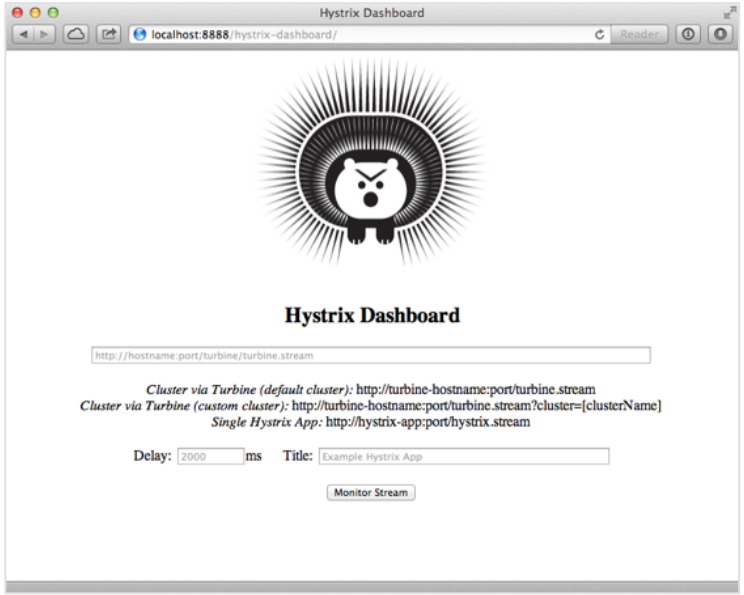


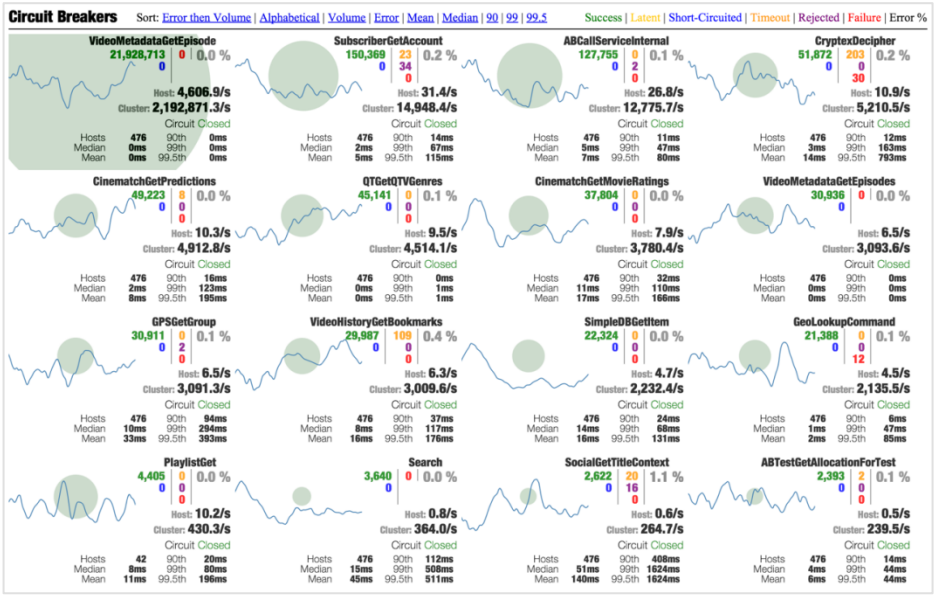
Rolling 10 second counters
with 1 second granularity

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

Hystrix Dashboard

- Hystrix dashboard is available at “/hystrix” endpoint and can be pointed to any “/hystrix.stream”





Hystrix Stream

- To embed Hystrix Dashboard in a spring cloud application
  - use dependecy

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

  - And add corresponding annotation to configuration class
    - `@EnableHystrixDashboard`

Lab 5

Lab 5 – Hystrix

- In this lab we will apply hystrix commands, fetch hystrix streams and watch the hystrix dashboard for metrics.

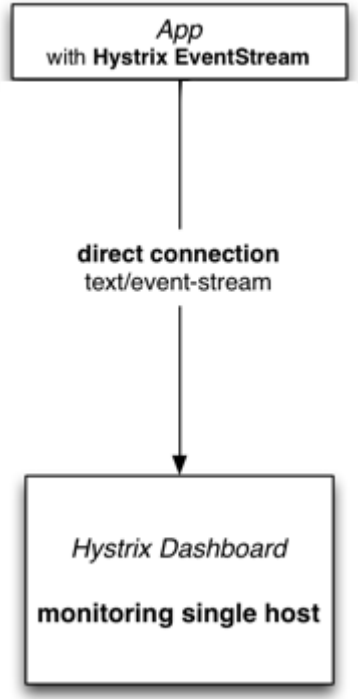
Turbine

Turbine

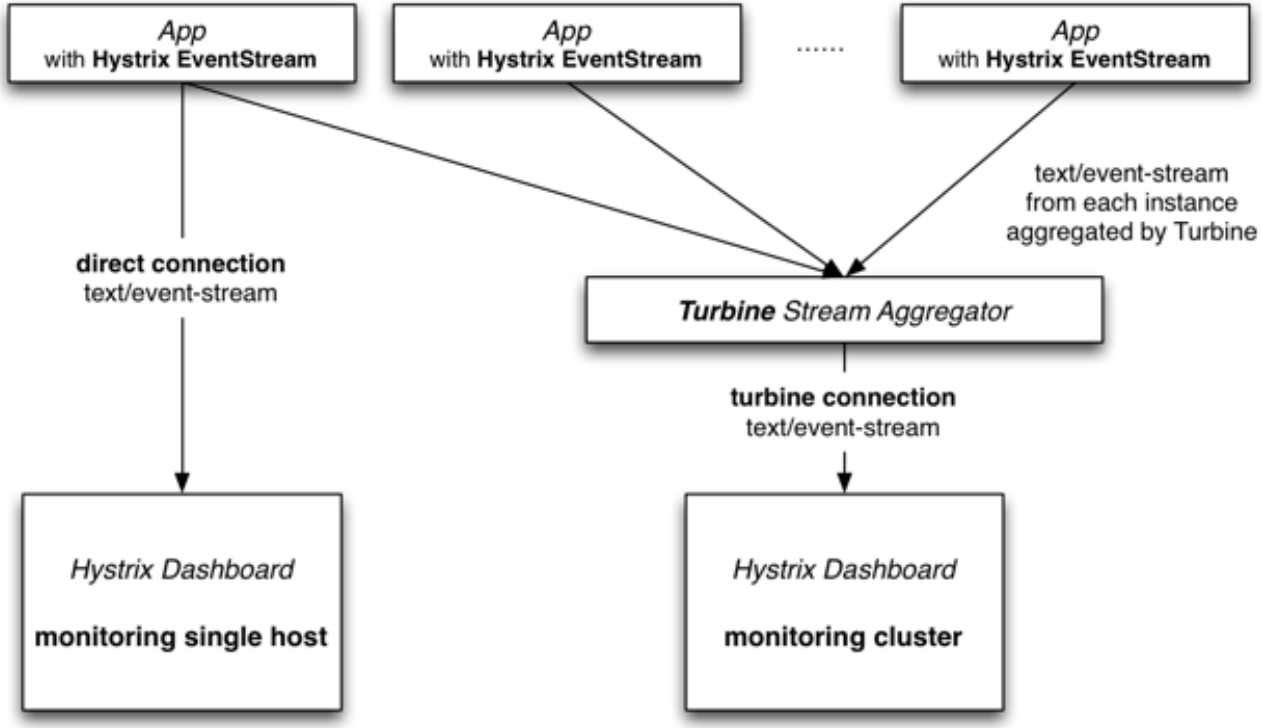
- Turbine is a tool for aggregating streams of Server-Sent Event (SSE) JSON data into a single stream.
- The targeted use case is metrics streams from instances in an SOA being aggregated for dashboards.
  - For example, Netflix uses Hystrix which has a realtime dashboard that uses Turbine to aggregate data from 100s or 1000s of machines.

Turbine

- Hystrix stream provides information on a single application
- However, in a microservices environment there may be more than one instances of an application online.
- Turbine is the component from Netflix that provides a way to aggregate this information across all installations of an application in a cluster.
- Turbine streams are Hystrix compatible and can be monitored using Hystrix Dashboard.

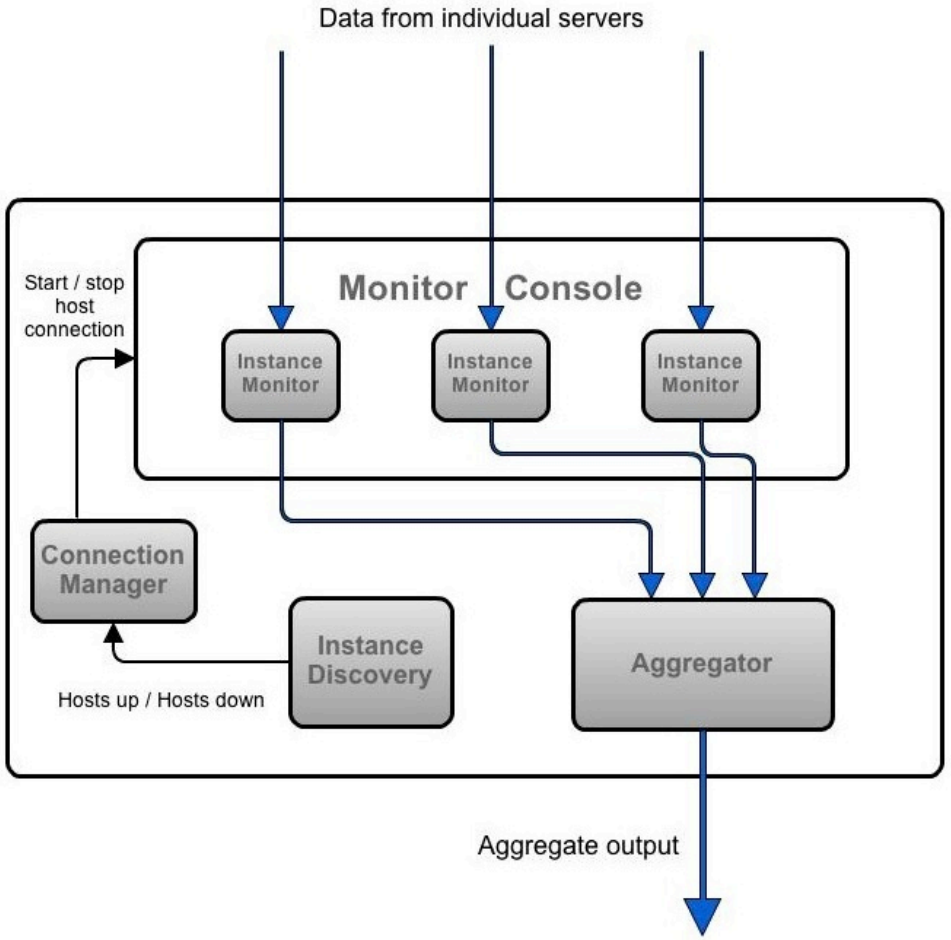


Turbine


App
with Hystrix EventStream
App
with Hystrix EventStream
.....
App
with Hystrix EventStream
text/event-stream
from each instance
aggregated by Turbine
direct connection
text/event-stream
Turbine Stream Aggregator
turbine connection
text/event-stream
Hystrix Dashboard
monitoring single host
Hystrix Dashboard
monitoring cluster

Turbine

- Architecture



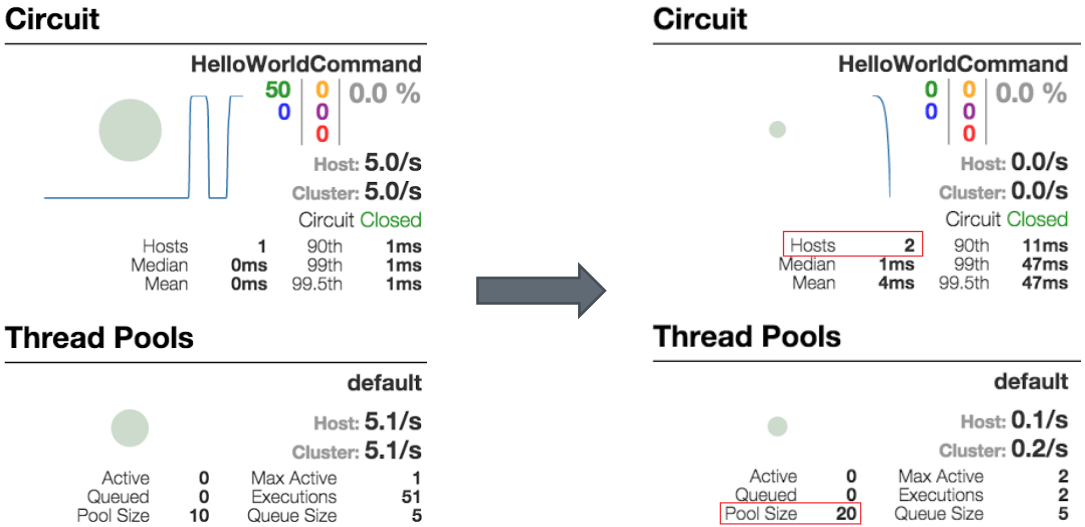
Turbine Stream

Turbine Stream

- To expose to /turbine.stream, Turbine
  - has to communicate with Eureka to find all available instances of a service;
  - aggregate available /hystrix.stream data from found instances to /turbine.stream;
- Turbine stream can be provided by any spring cloud application, that has access to Eureka.

Turbine Stream

- If a cluster of two instances is aggregated by Turbine, the counts against the host indicate the two hosts for which the stream is being aggregated and thread pool size count indicate sum of the pools from the two hosts;



Turbine Stream

- To utilize Turbine
  - Add turbine starter dependency with exclusion of servlet-api to avoid startup issues;

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine</artifactId>
    <exclusions>
        <exclusion>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

  - Add @EnableTurbine annotation to application configuration class

Turbine Stream

- To utilize Turbine (continuation)
  - Add the turbine cluster configuration and make sure turbine application has read access to Eureka;

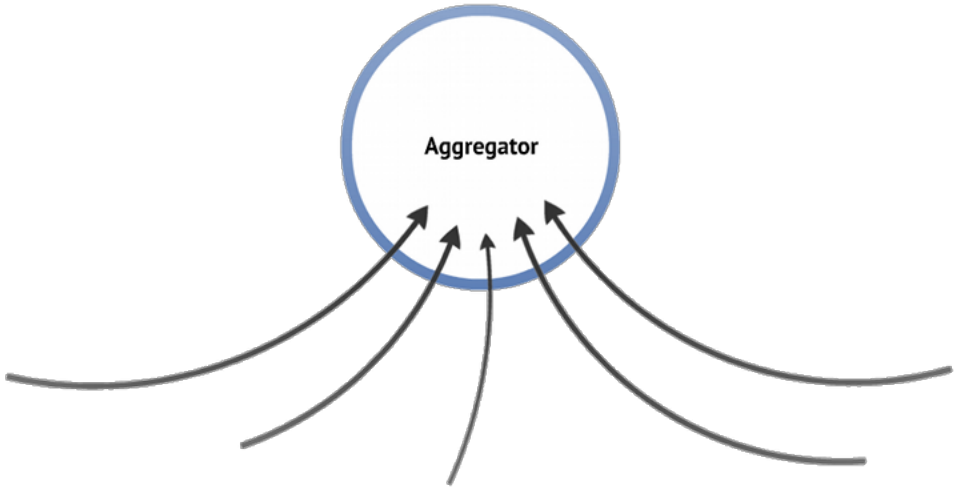
```
turbine:
  aggregator:
    clusterConfig: SERVICE1,SERVICE2
  appConfig: Service1,Service2
```

- Depending on the nature of turbine eureka provider plugin, **clusterConfig** and **appConfig** should match, where **clusterConfig** is the uppercase app name from Eureka and **appConfig** is case-insensitive name of the spring application;

Lab 6

Lab 6 – Turbine

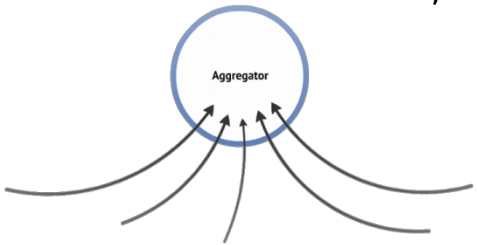
- In this lab we will aggregate hystrix streams from several instances of a microservice into a single turbine stream.



Turbine AMQP

Turbine AMQP

- Turbine provides a way to aggregate the information from Hystrix streams across a cluster.
  - Cluster is restricted to instances of a single service;
    - What to do, if it is required to monitor all microservices at once?
  - Services are restricted to only those, that are registered at discovery server;
    - What to do, if the application we want to monitor, is not registered at discovery server?

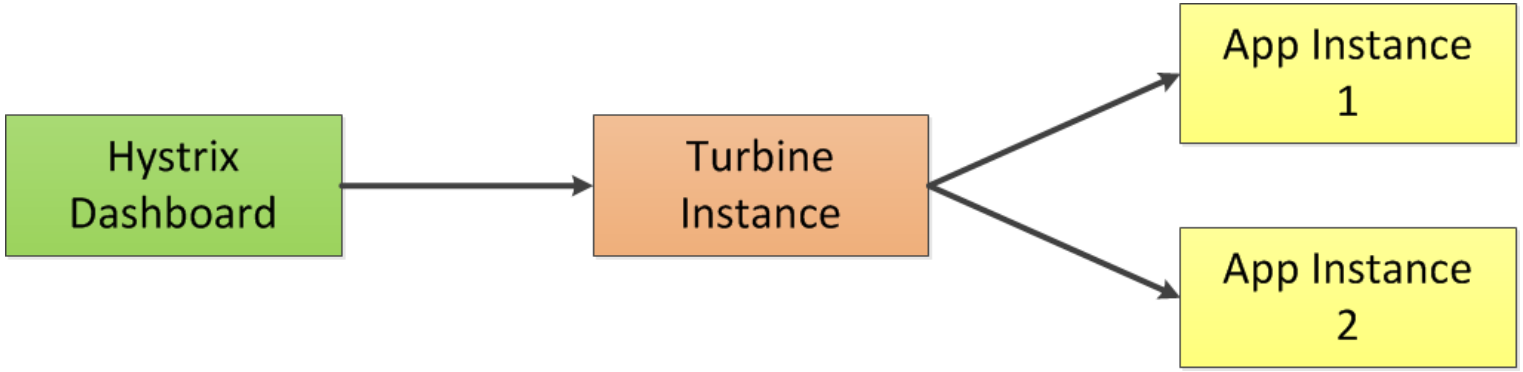


Turbine AMQP

- The default aggregation flow is pull-based, where Turbine requests the hystrix stream from each instance in the cluster and aggregates it together.
  - This tends to be way more configuration heavy;
    - Configuration files should be maintained with every new microservice
  - In some cases this is just not achievable due to technical restrictions;
    - With some containerized environments turbine may not be successful to reach remote hystrix streams;

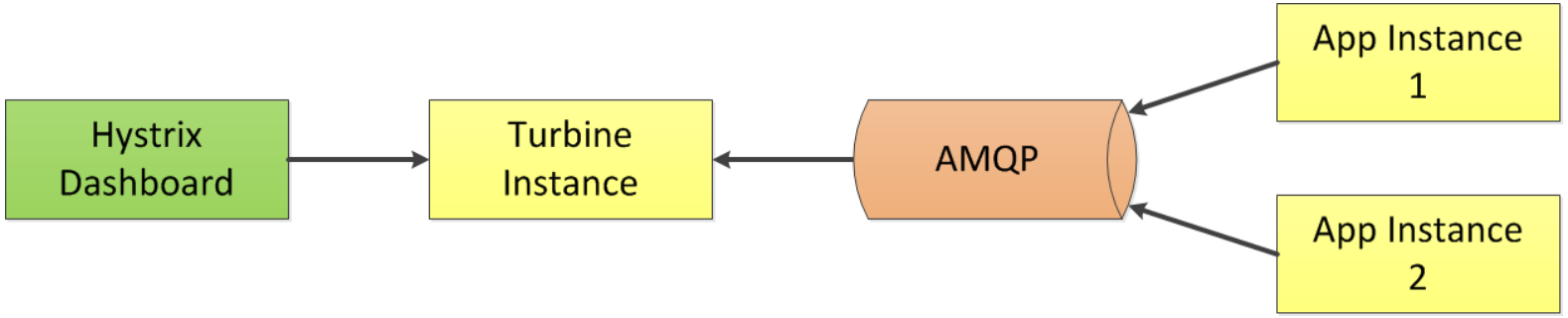
Turbine AMQP

- Turbine default aggregation flow



Turbine AMQP

- Spring Cloud Turbine AMQP offers a different model
  - where each application instance pushes the metrics from Hystrix commands to Turbine through a central AMQP broker.



Turbine AMQP

- With minor changes for such a powerful feature, an application which wants to feed the hystrix stream to an AMQP broker should add a few dependencies expressed in maven the following way:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

  - These dependencies would auto-configure all the connectivity details with RabbitMQ topic exchange and would start feeding in the hystrix stream data into the RabbitMQ topic.

Turbine AMQP

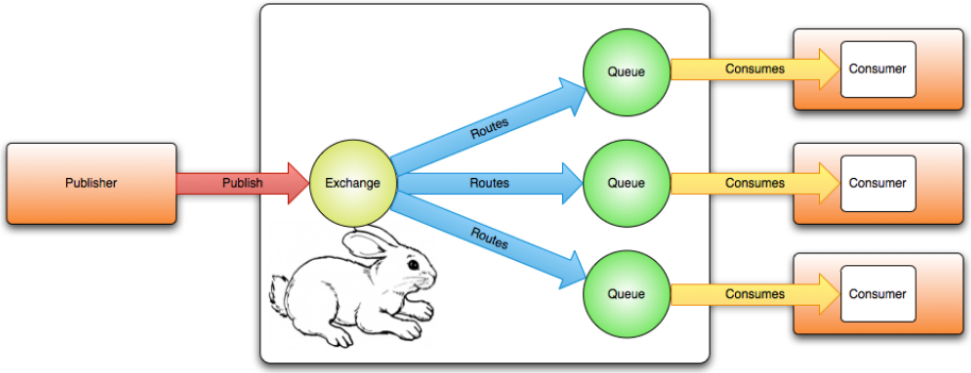
- Similarly on the Turbine end all that needs to be done is to specify

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

  - And use **@EnableTurbineStream** annotation in configuration class
    - This would consume the hystrix messages from RabbitMQ and would in turn expose an aggregated stream over an http endpoint.

Turbine AMQP

- Spring Cloud Turbine AMPQ dependencies will automatically set default configuration values for message broker and queue.
  - RabbitMQ exchange name is
    - springCloudHystrixStream
  - RabbitMQ connection settings are controlled with the following properties:
    - spring.rabbitmq.host: localhost
    - spring.rabbitmq.port: 5672
    - spring.rabbitmq.username: guest
    - spring.rabbitmq.password: guest



Lab 7

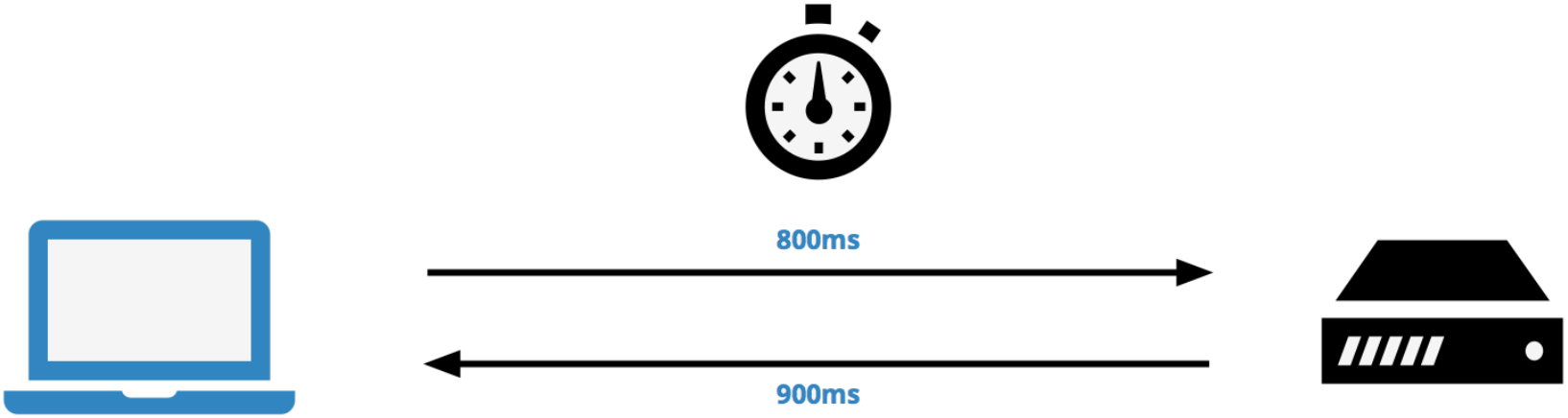
Lab 7 – Turbine AMQP

- In this lab we will aggregate different hystrix streams into a single turbine stream.

Troubleshooting Latency Issues

Troubleshooting Latency Issues

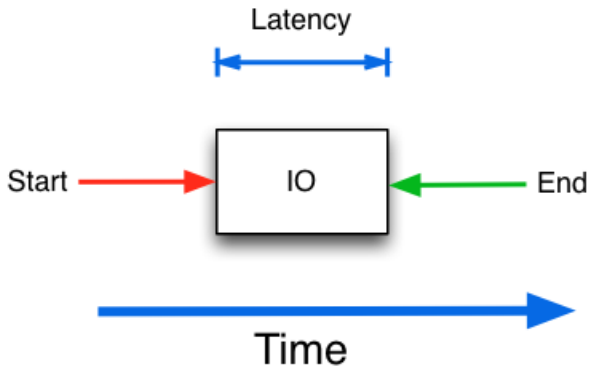
- What is Latency?



Latency = 800ms + 900ms = 1.7s

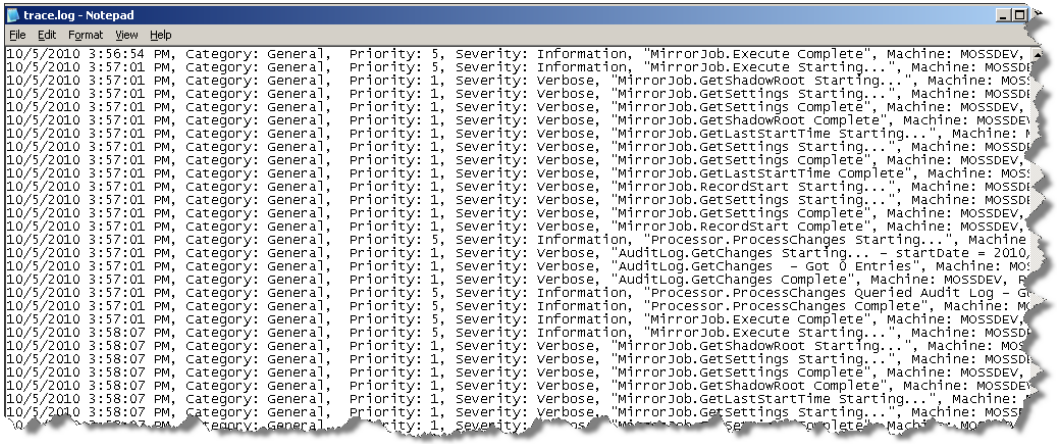
Troubleshooting Latency Issues

- Whenever a latency issue happens, we want to know
  - When was the event and how long did it take?
  - Where did this happen?
  - Which event was it?
  - Is it abnormal?



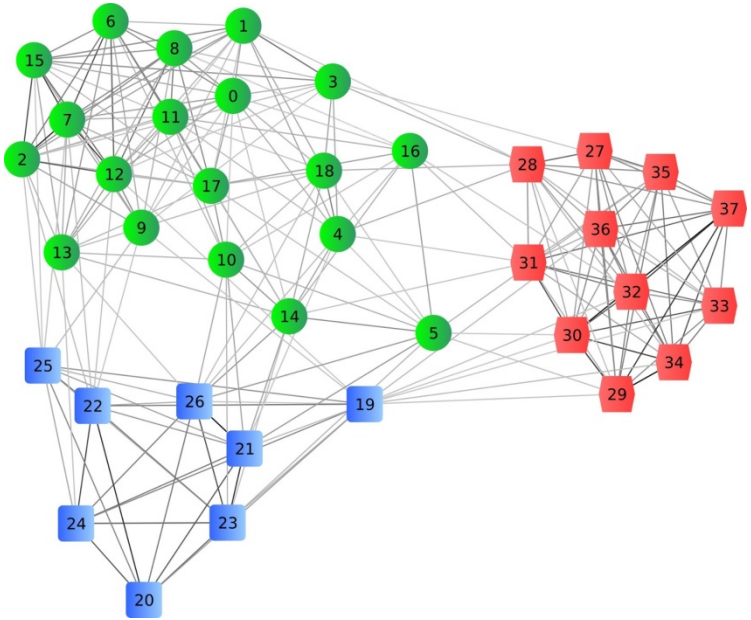
Troubleshooting Latency Issues

- When was the event and how long did it take?
  - Which request?
  - What time?
  - Which line in log?



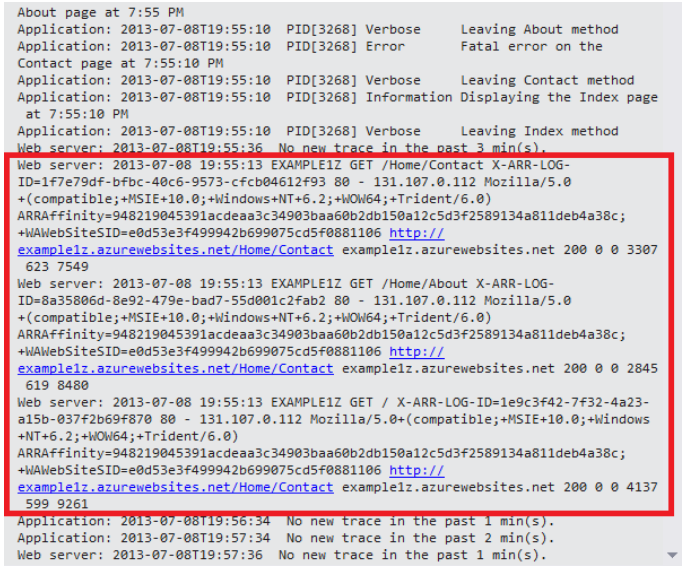
Troubleshooting Latency Issues

- Where did this happen?
  - Which cluster?
  - Which shard?
  - Which server?



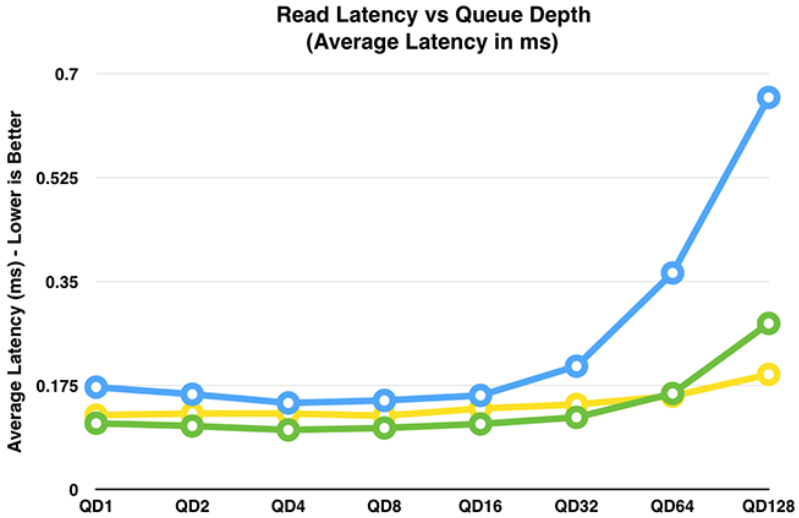
Troubleshooting Latency Issues

- Which event was it?
  - Which exact request?
  - Which request id?



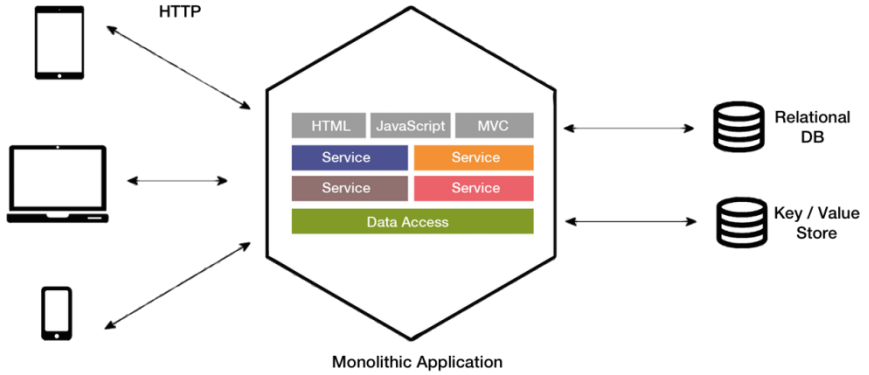
Troubleshooting Latency Issues

- Is it abnormal?
  - What is the average latency for this event?



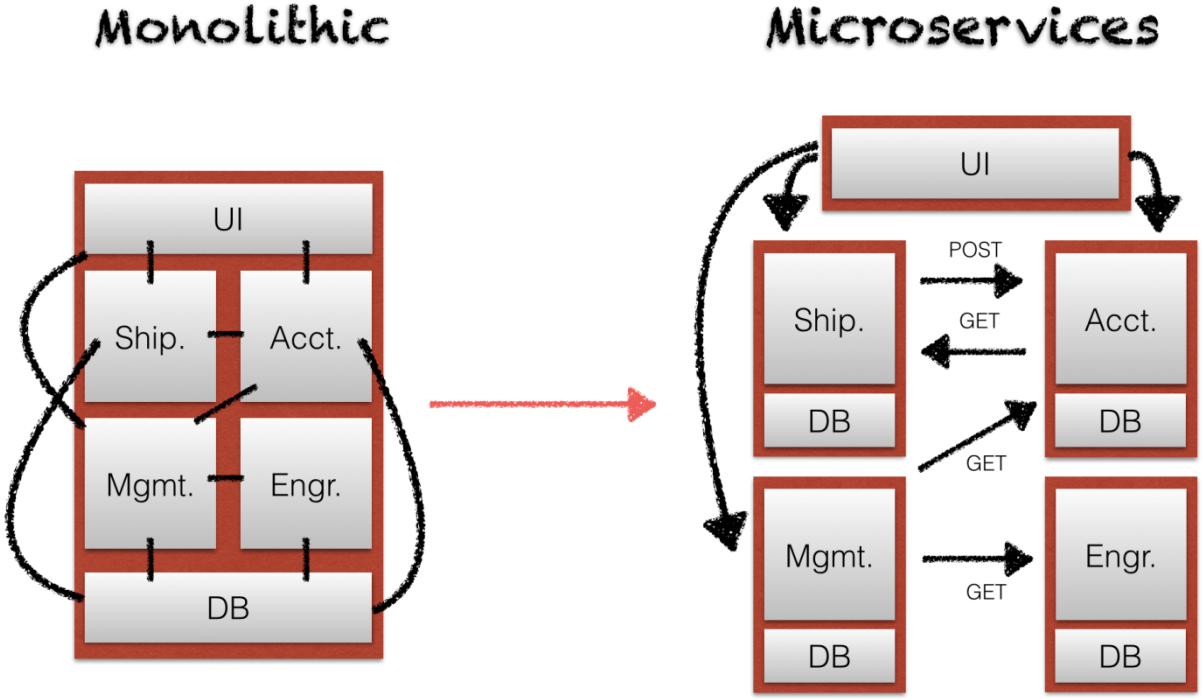
Troubleshooting Latency Issues

- When all of an application's functionality lives in a *monolith* it's much easier to reason about where things have gone wrong.
  - *monolith* – is what we call applications written as one, large, unbroken deployable like a .war or .ear



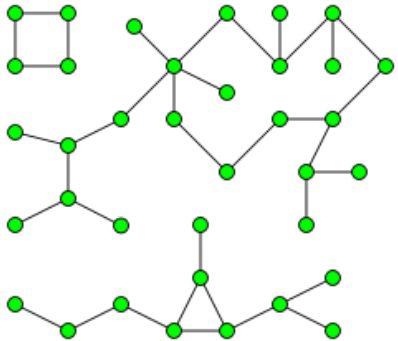
Troubleshooting Latency Issues

- Distribution changes everything.



Troubleshooting Latency Issues

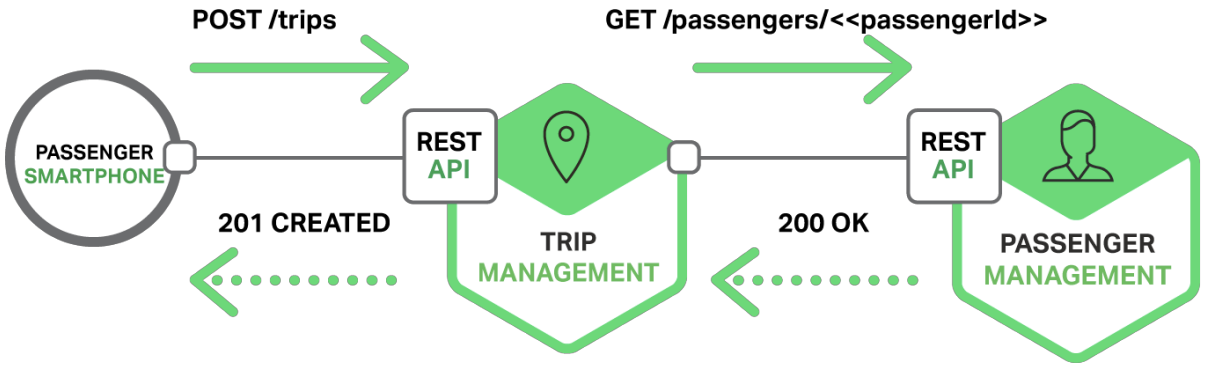
- Microservice architectures are similar to a graph of components, distributed across a network
  - A call graph can become delayed or fail due to the nature of the operation, components, or edges between them.
  - We want to understand our current architecture and troubleshoot latency issues, in production.



Distributed Tracing

Distributed Tracing

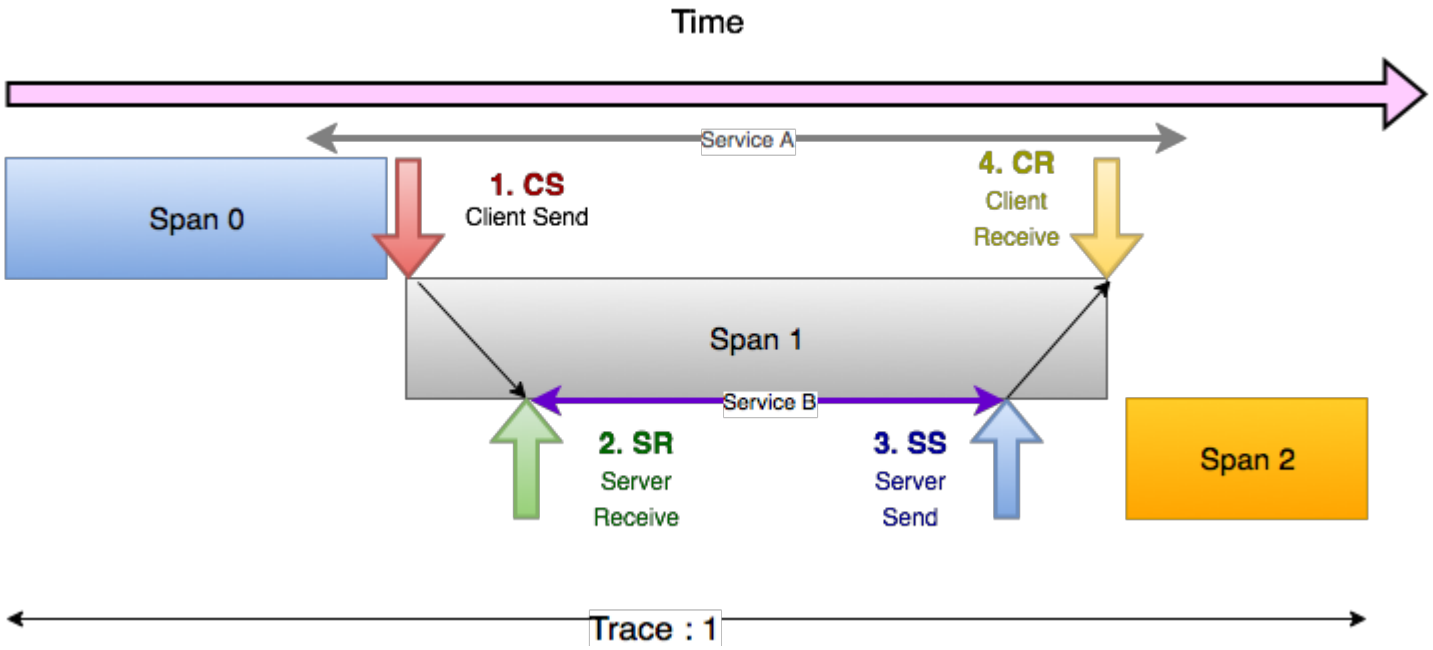
- Distributed tracing systems collect end-to-end latency graphs (traces) in near real-time.
- You can compare traces to understand why certain requests take longer than others.



Distributed Tracing

- As a request flows from one component to another in a system, through ingress and egress points, **tracers** add logic where possible to perpetuate a unique **trace ID** that's generated when the first request is made.
  - As a request arrives at a component along its journey, a new **span ID** is assigned for that component and added to the trace.

Distributed Tracing



Distributed Tracing

- A trace represents the whole journey of a request, and a span is each individual hop along the way, each request.
  - Spans may contain **tags**, or metadata, that can be used to later contextualize the request.
  - Spans typically contain common tags like start timestamps and stop timestamp, though it's easy to associate semantically relevant tags like an a business entity ID with a span.

Distributed Tracing

- A Span is the basic unit of work.
  - For example, sending an RPC is a new span, as is sending a response to an RPC.
- Span’s are identified by a unique 64-bit ID for the span and another 64-bit ID for the trace the span is a part of.
  - Spans also have other data, such as descriptions, key-value annotations, the ID of the span that caused them, process ID’s (normally IP address), and they keep track of their timing information.

Distributed Tracing

- Distributed tracing support is provided by Spring Cloud Sleuth.
  - Adds trace and span ids to the Slf4J MDC, so you can extract all the logs from a given trace or span in a log aggregator.
  - Provides an abstraction over common distributed tracing data models: traces, spans (forming a DAG), annotations, key-value annotations.
    - Loosely based on HTrace, and Zipkin (Dapper) compatible.
  - Instruments common ingress and egress points from Spring applications.
    - servlet filter, rest template, scheduled actions, message channels, zuul filters, feign client.

Distributed Tracing

- How Much Data is Enough?
  - Which requests should be traced?
- Ideally, you'll want enough data to see trends reflective of live, operational traffic.
  - You don't want to overwhelm your logging and analysis infrastructure, though.

Distributed Tracing

- Some organizations may only keep requests for every thousand requests, or every ten, or every million!
- By default, the threshold is 10%, or .1, though you may override it by specifying a sampling percentage:
  - spring.sleuth.sampler.percentage: 0.2

Distributed Tracing

- Make sure to set realistic expectations for your application and infrastructure.
  - It may well be that the usage patterns for your applications require something more sensitive or less sensitive to detect trends and patterns.
  - This is meant to be operational data; most organizations don't warehouse this data more than a few days or, at the upper bound, a week.

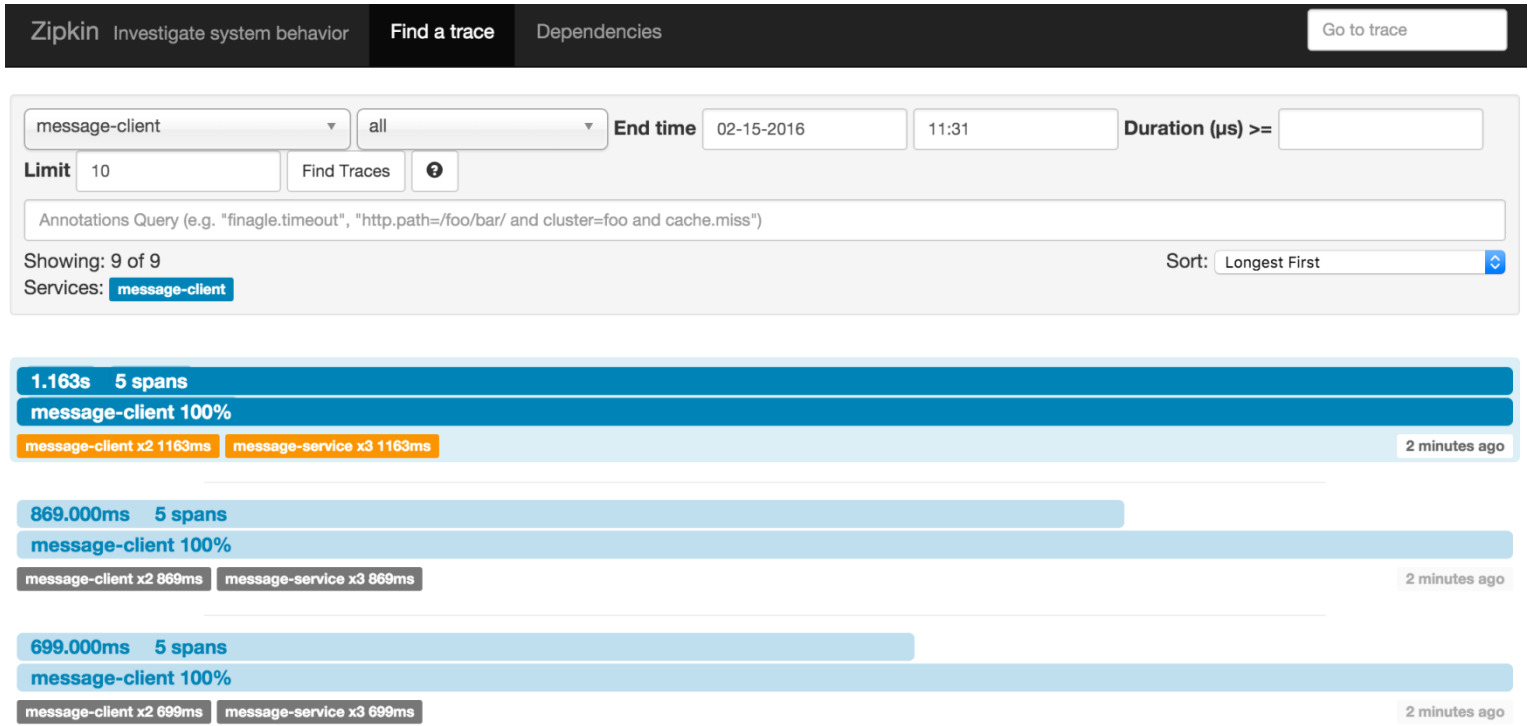
Zipkin

Zipkin

- Data collection is a start but the goal is to *understand* the data, not just collect it. In order to appreciate the big picture, we need to get beyond individual events.
- For this we'll use the OpenZipkin project. OpenZipkin is the fully open-source version of Zipkin, a project that originated at Twitter in 2010, and is based on the Google Dapper papers.

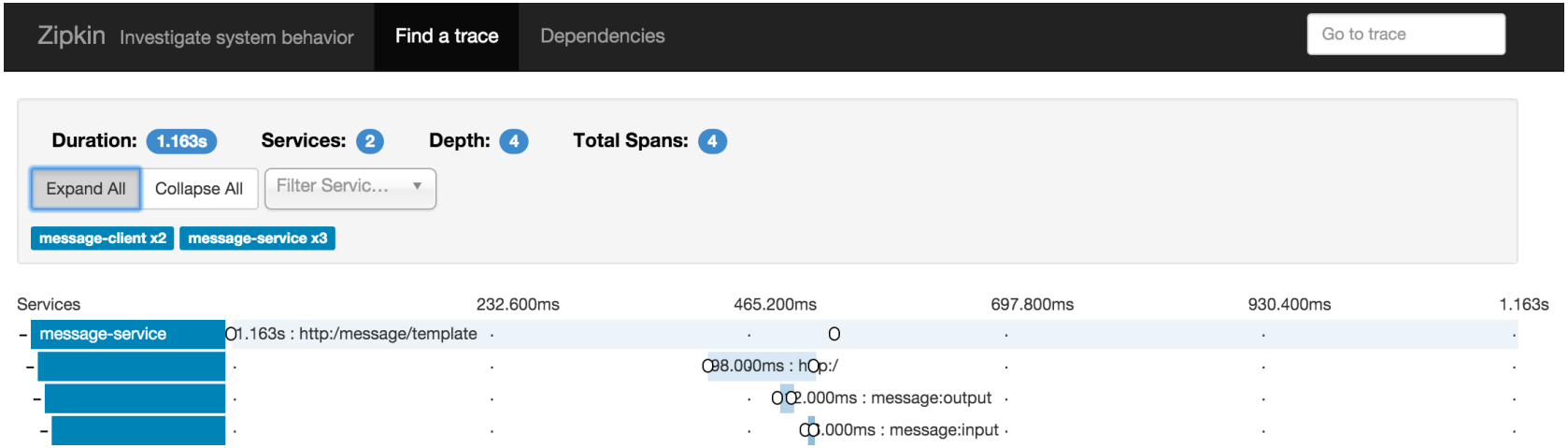
Zipkin

- The Zipkin web UI visualizes the traces according to filters set.



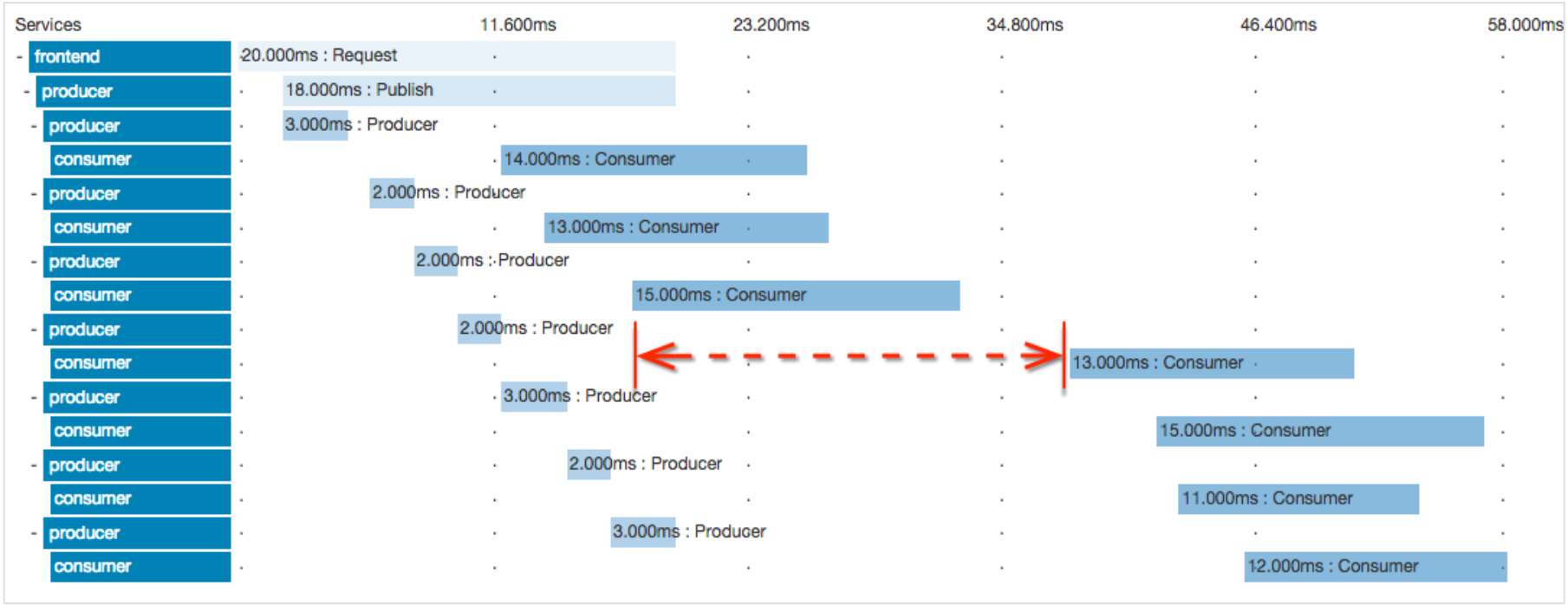
Zipkin

- When a trace is clicked, spans are visualized.



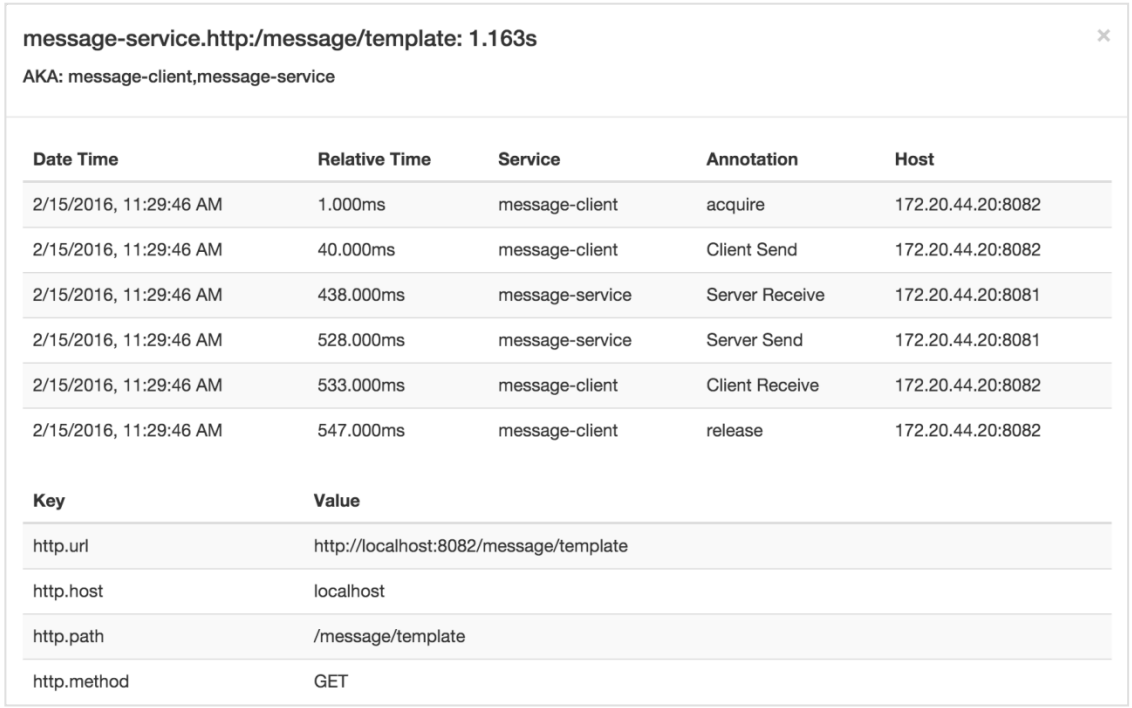
Zipkin

- When a trace is clicked, spans are visualized.



Zipkin

- Each individual span also carries information about the particular request with which its associated. You can view this detail by clicking on an individual span:



message-service.http:/message/template: 1.163s

AKA: message-client,message-service

Date Time	Relative Time	Service	Annotation	Host
2/15/2016, 11:29:46 AM	1.000ms	message-client	acquire	172.20.44.20:8082
2/15/2016, 11:29:46 AM	40.000ms	message-client	Client Send	172.20.44.20:8082
2/15/2016, 11:29:46 AM	438.000ms	message-service	Server Receive	172.20.44.20:8081
2/15/2016, 11:29:46 AM	528.000ms	message-service	Server Send	172.20.44.20:8081
2/15/2016, 11:29:46 AM	533.000ms	message-client	Client Receive	172.20.44.20:8082
2/15/2016, 11:29:46 AM	547.000ms	message-client	release	172.20.44.20:8082

Key	Value
http.url	http://localhost:8082/message/template
http.host	localhost
http.path	/message/template
http.method	GET

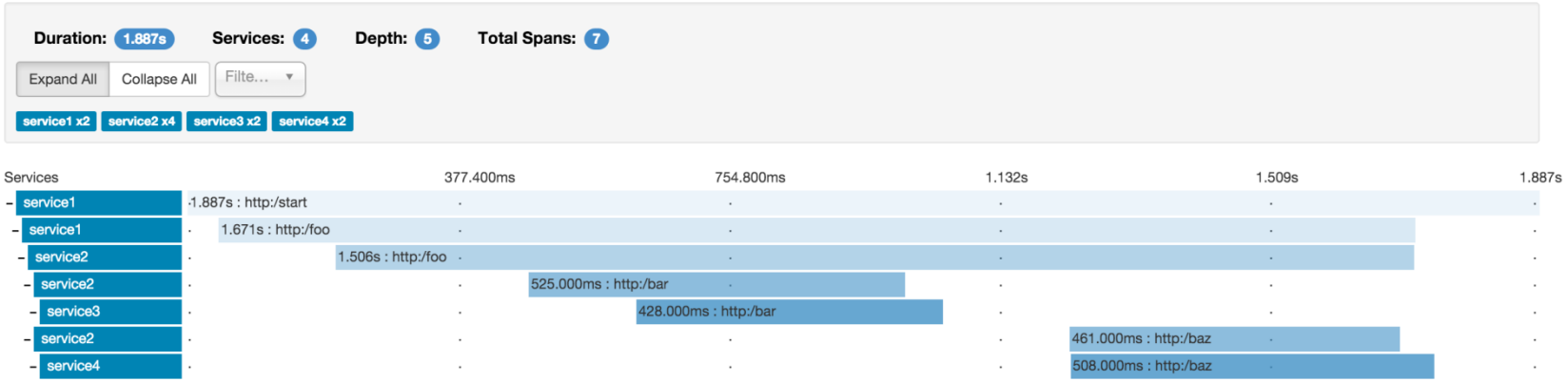
Zipkin

- Any application to be instrumented with Sleuth and Zipkin should have dependency to **`spring-cloud-sleuth-stream`**
  - As well as corresponding stream dependency (RabbitMQ, Kafka)
- The downside is that Sleuth will wrap all of Hystrix stream and other annoying things, however, all of them could be filtered not to expose them to Zipkin server.
  - You could read more about these features at Sleuth documentation at https://cloud.spring.io/spring-cloud-sleuth/spring-cloud-sleuth.html

Lab 8

Lab 8 – Distributed Tracing with Zipkin

- In this lab we will instrument services to provide tracing data available to Zipkin.



Thank You!

LXFT
LISTED
NYSE

LUXOFT