# Spring Cloud Hystrix

Understanding and Applying
Client Side Circuit Breakers

# Objectives

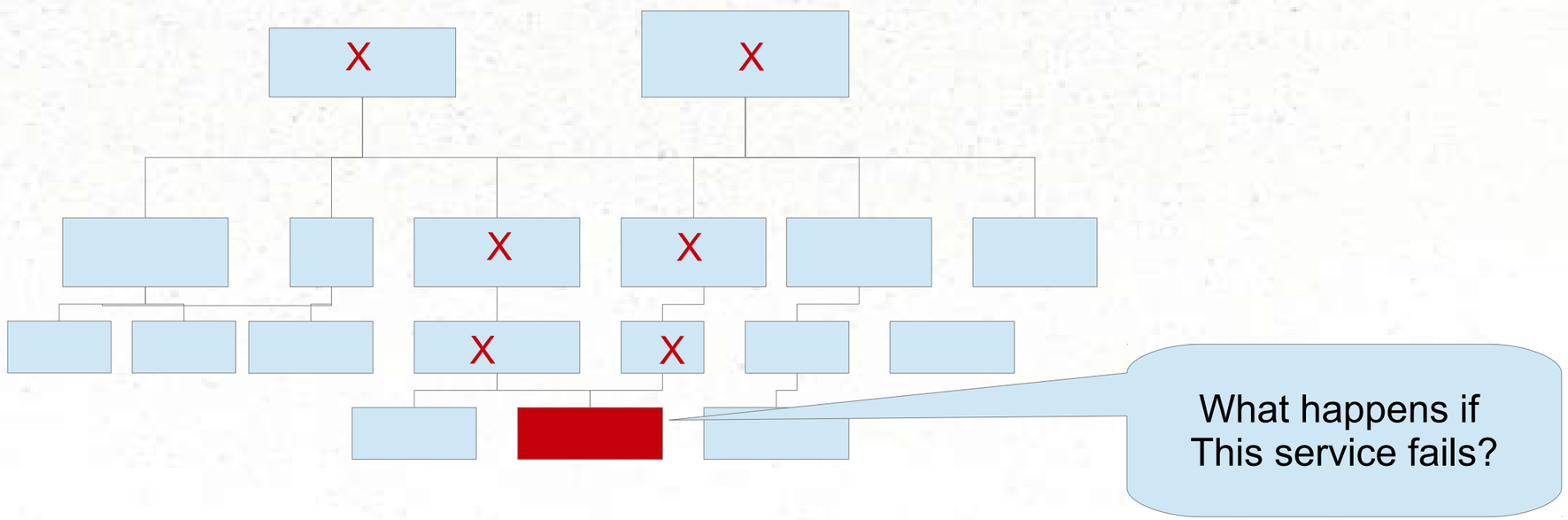
- At the end of this module, you will be able to
  - Understand how software circuit breakers protect against cascade failure
  - Use Spring Cloud Netflix Hystrix annotations within your software to implement circuit breakers
  - Establish simple monitoring of Circuit Breakers using Hystrix Dashboard and Turbine

# Module Outline

- **Cascading Failures and the Circuit Breaker Solution**
- Using Spring Cloud Netflix Hystrix
- Monitoring with the Hystrix Dashboard and Turbine

# The Problem: Cascading Failure

- Having a large number of services as dependencies can lead to a 'cascading failures'
- Without mitigating this, microservices are a recipe for certain disaster!



# Distributed Systems –
## More Failure Opportunities

- Distributed systems → more opportunity for failure.
  - Remember the Fallacies of Distributed Computing.
- The Math: Assume 99.95% Uptime (Amazon EC2 SLA)
  - Single app – 22 minutes down per month
  - 30 interrelated services – 11 hours downtime per month (bad)
  - 100 interrelated services – 36 hours downtime per month (ouch!)

# The Circuit Breaker Pattern

- Consider a household circuit breaker
  - It “watches” a circuit
  - When failure occurs (too much current flow), it “opens” the circuit (disconnects the circuit)
  - Once problem is resolved, you can manually “close” the breaker by flipping the switch.
  - Prevents cascade failure
    - i.e. - your house burning down.

# Module Outline

- Cascading Failures and the Circuit Breaker Solution
- **Using Spring Cloud Netflix Hystrix**
- Monitoring with the Hystrix Dashboard and Turbine

# Hystrix – The Software Circuit Breaker

- Hystrix – Part of Netflix OSS
- Light, easy-to-use wrapper provided by Spring Cloud.
- Detects failure conditions and “opens” to disallows further calls
  - Hystrix Default – 20 failures in 5 seconds
- Identify “fallback” - what to do in case of a service dependency failure
  - Think: catch block, but more sophisticated
  - Fallbacks can be chained
- Automatically “closes” itself after interval
  - Hystrix Default – 5 seconds.

# Comparison with Physical Circuit Breaker

| HYSTRIX<br>DEFEND YOUR APP |  |
|---|---|
| • “closed” when operating normally<br>• “open” when failure is detected<br>• Failure definition flexible<br>– Exception thrown or timeout exceeded over time period<br>• Definable “Fallback” option<br>• Automatically re-closes itself | • “closed” when operating normally<br>• “open” when failure is detected<br>• Failure definition fixed<br>– current flow exceeds amp rating.<br>• No fallback<br>• Must be closed manually |

# Hystrix (Spring Cloud) Setup

- Add the Dependency:

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

- Enable Hystrix within a configuration class:

```
@SpringBootApplication
@EnableHystrix
public class Application {
}
```

# Hystrix (Spring Cloud) Example

- Use the @HystrixCommand to wrap methods in a circuit breaker:

Based on recent failures,
Hystrix will call
one of these two methods

```
@Component
public class StoreIntegration {

    @HystrixCommand(fallbackMethod = "defaultStores")
    public Object getStores(Map<String, Object> parameters) {
        //do stuff that might fail
    }

    public Object defaultStores(Map<String, Object> parameters) {
        return /* something useful */;
    }
}
```

# Custom Properties

- Failure / Recovery behavior highly customizeable
  - Use commandProperties and @HystrixProperty

Over 20% failure rate in 10 second period, open breaker

```
@HystrixCommand(
 fallbackMethod = "defaultStores",
 commandProperties = {
  @HystrixProperty( name="circuitBreaker.errorThresholdPercentage", value="20"),
  @HystrixProperty( name="circuitBreaker.sleepWindowInMilliseconds", value="1000")
  })
public Object yourMethod( … ) {
  // …
}
```

After 1 second, try closing breaker

See: Netflix Hystrix – Hystrix Javanica Configuration
https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#configuration

# Command can be called various ways

- **Synchronously** – call execute and block thread (default behavior).

- **Asynchronously** – Call in a separate thread (queue), returning a future. Deal with `Future` when you want
  - Just like Spring's @Async annotation

- **Reactively** – Subscribe, get a listener (`Observable`)

# Example: Asynchronous Command Execution

- Have method return Future
  - Wrap result in `AsyncResult`

```
@HystrixCommand( … )
public Future<Store> getStores(Map<String, Object> parameters) {
   return new AsyncResult<Store>() {
      @Override
      public Store invoke() {
         //do stuff that might fail
      }
   };
}
```

# Example: Reactive Command Execution

- Have method return Observable
  - Wrap result in `ObservableResult`

```
@HystrixCommand( … )
public Observable<Store> getStores(Map<String, Object> parameters) {
   return new ObservableResult<Store>() {
      @Override
      public Store invoke() {
         //do stuff that might fail
      }
   };
}
```

# Hystrix Properties

- *execution.isolation.thread.**timeout**InMilliseconds*

  How long should we wait for success?
- *circuitBreaker.requestVolumeThreshold*

  # of requests in rolling time window (10 seconds) that activate the circuit breaker (NOT the # of errors that will trip the breaker!)
- *circuitBreaker.errorThresholdPercentage*

  % of failed requests that will trip the breaker (default = 50%)
- *metrics.rollingStats.timeInMilliseconds*

  Size of the rolling time window (default = 10 seconds)

# How to Reset the Circuit Breaker?

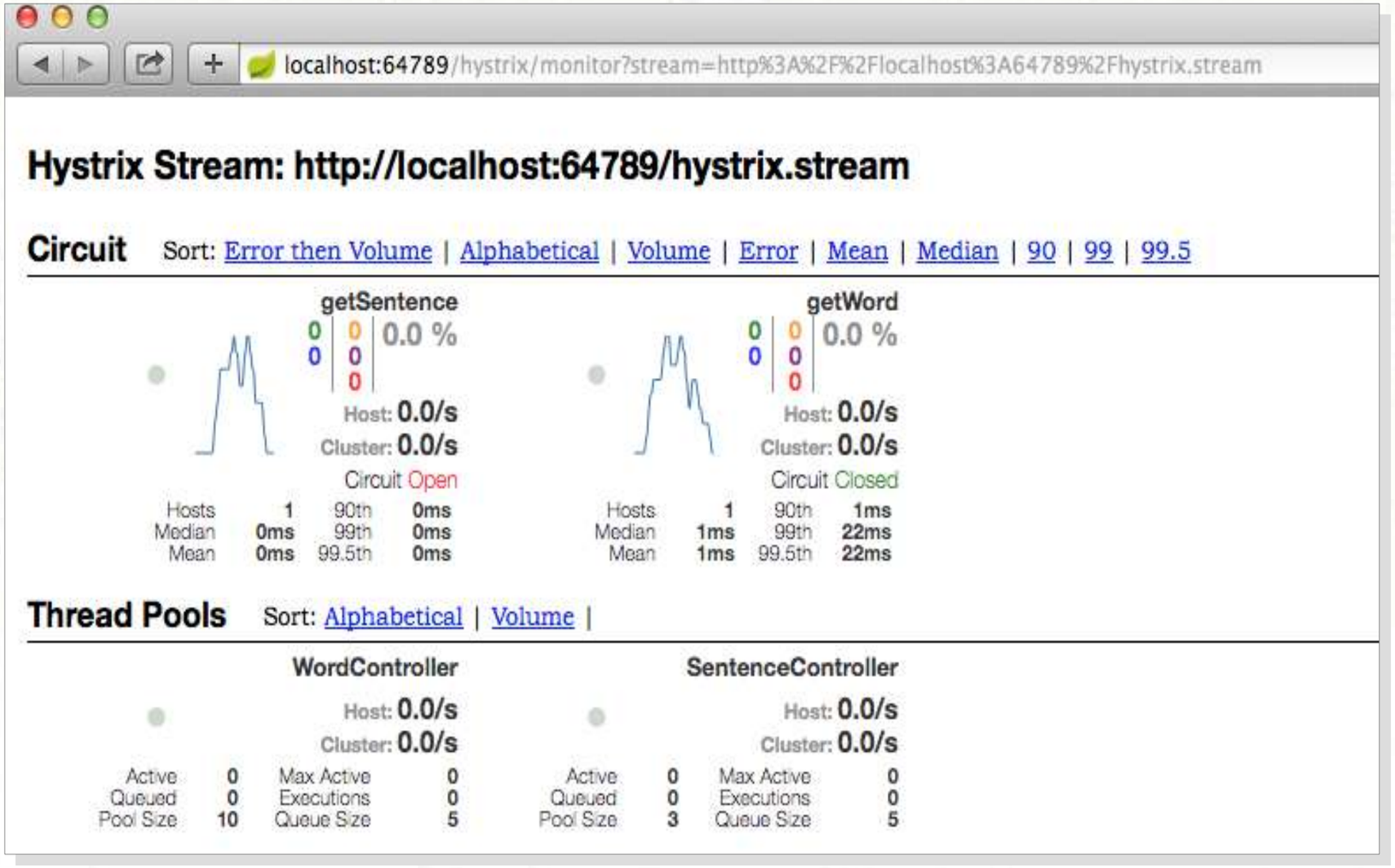
- When the failing service is healthy, we want to 'close' the circuit breaker again
- *circuitBreaker.sleepWindowInMilliseconds*
  - How long to wait before closing the breaker (default = 5 seconds)
- *circuitBreaker.forceClosed*
  - Manually force the circuit breaker closed

# Module Outline

- Cascading Failures and the Circuit Breaker Solution
- Using Spring Cloud Netflix Hystrix
- **Monitoring with the Hystrix Dashboard and Turbine**

# Hystrix Dashboard

- Hystrix provides a built-in dashboard to check the status of the circuit breakers:



# Hystrix Dashboard Setup

- Add the additional Dependency, include actuator:

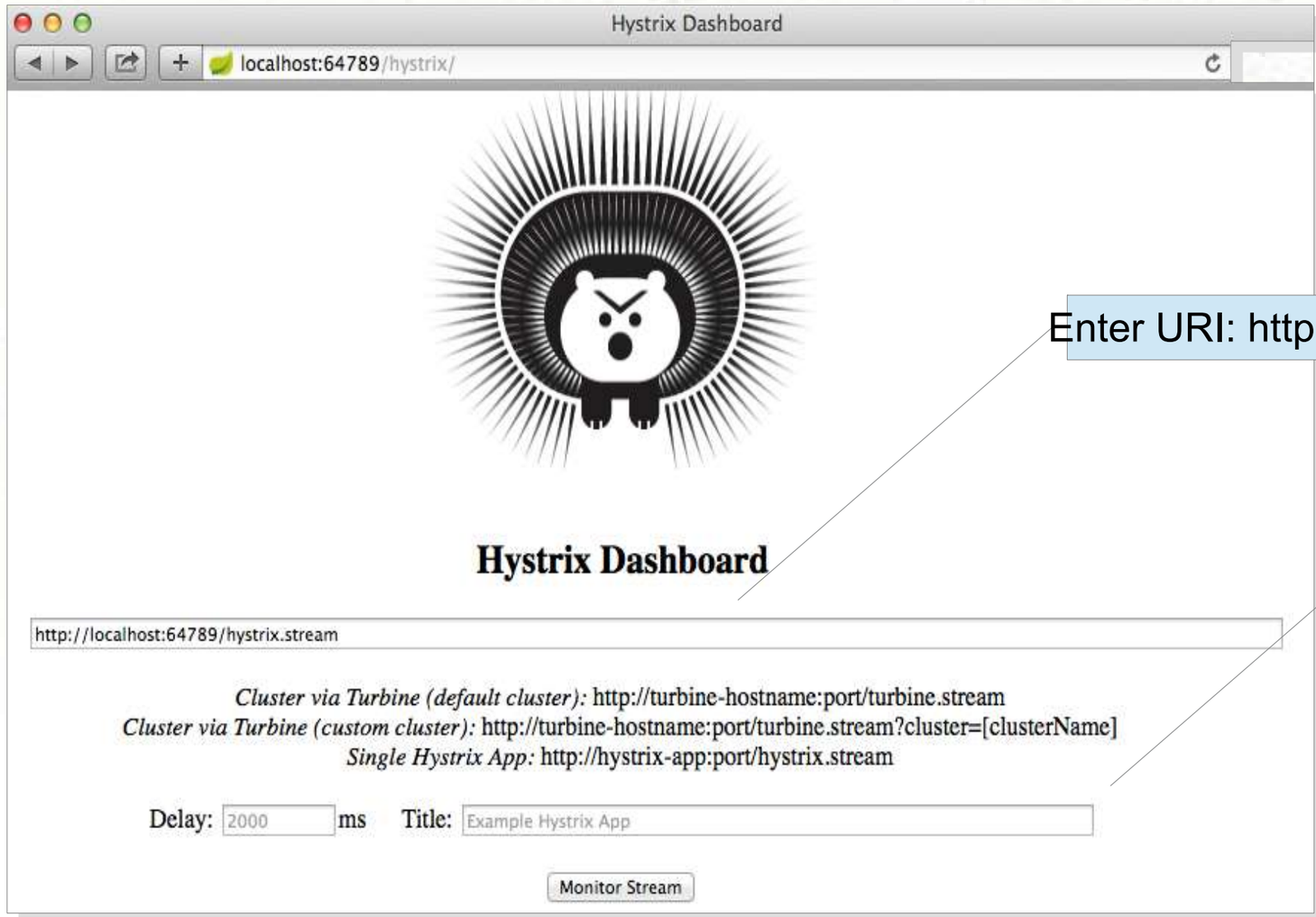
```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- Enable Hystrix Dashboard within a configuration class:

```
@SpringBootApplication
@EnableHystrix
@EnableHystrixDashboard
public class Application {
}
```

# Accessing Hystrix Dashboard

- Use URI: http://<host>:<port>/hystrix

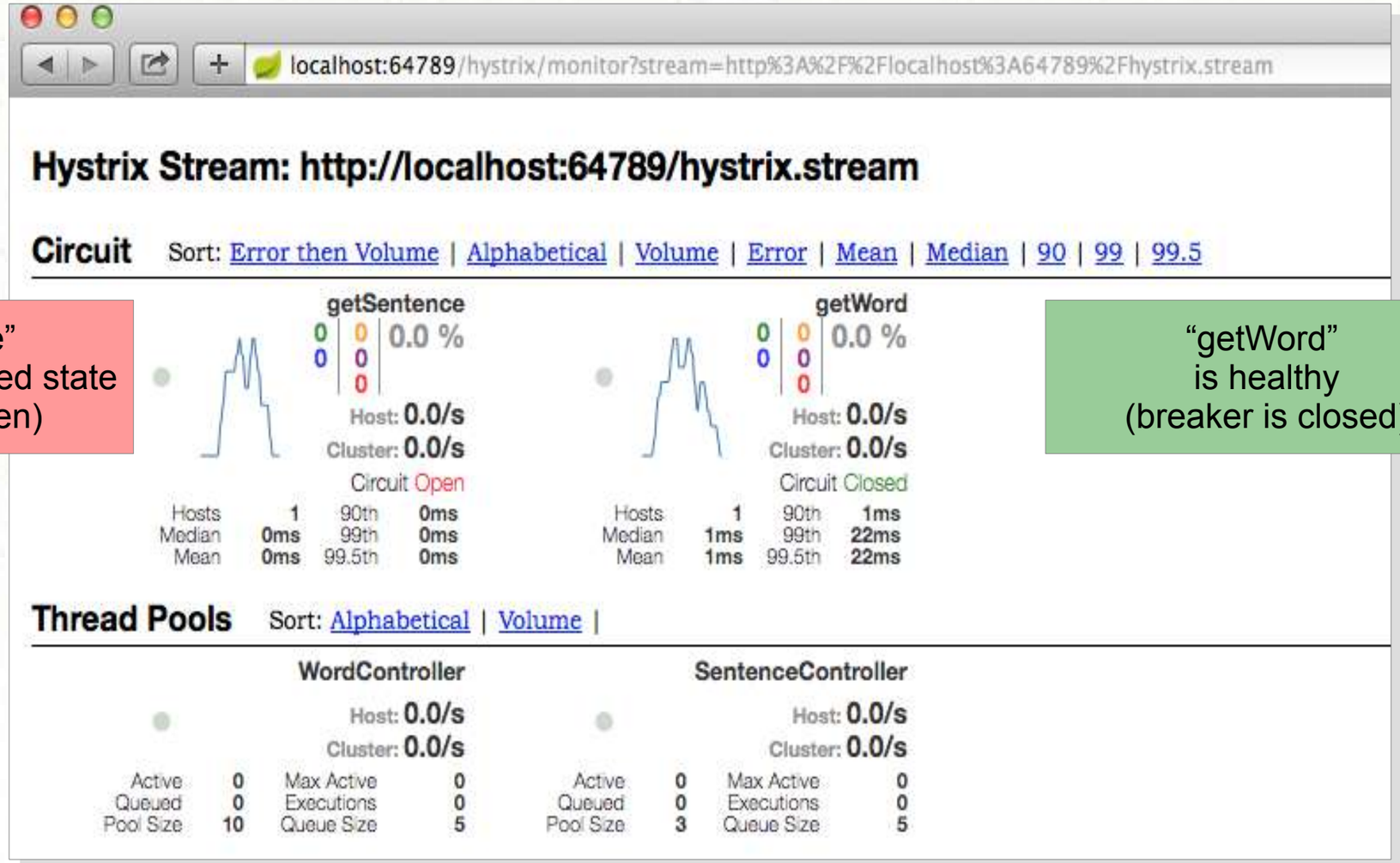



Enter URI: http://<host>:<port>/hystrix.**stream**

Enter whatever title you want to see on the screen

# The Hystrix Dashboard

- Each method annotated with @HystrixCommand appears on the dashboard:

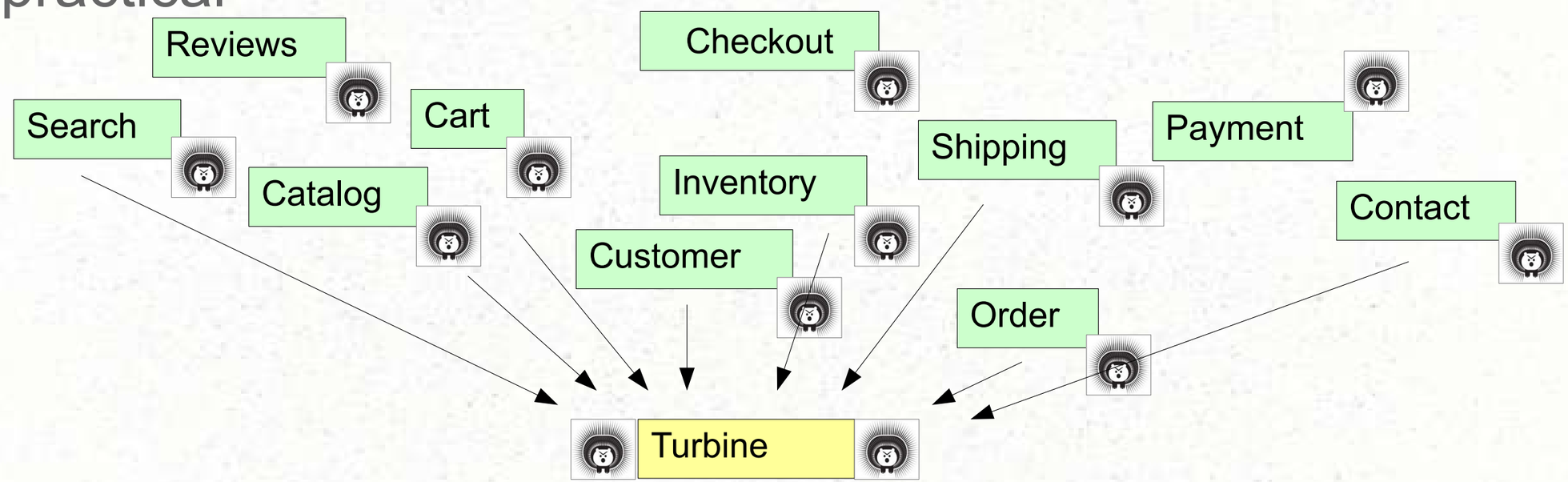



"getSentence"
Is currently in a failed state
(breaker is open)

"getWord"
is healthy
(breaker is closed)

# Monitoring Using Turbine

- Monitoring large numbers of Hystrix dashboards isn't really practical




- Turbine provides a consolidated view
  - Gathers metrics from the individual instances

# Summary

- Software circuit breakers protect against cascade failure
- Spring Cloud Netflix Hystrix provides an easy way to add circuit breakers to your applications
- You can use Hystrix Dashboard and Turbine to monitor your circuit breakers.

# Exercise

Instructions: Student Files, Lab 7