Application Hardening with Netflix Hystrix

Objectives

Key objectives of this chapter

- Netflix Hystrix
- Patterns
- Circuit Breaker Configuration
- Fallback Configuration
- Collapser Configuration
- The Monitor

1.1 Netflix Hystrix

- Hystrix was designed by Netflix to run in the Amazon AWS Cloud
- Designed to :
  - Handle latency across service
  - Handle failures across services
  - Stop cascading failures
  - Provide fallbacks
  - Improve system resiliency
- Hystrix is named after a kind of endangered porcupine

HYSTRIX
DEFEND YOUR APP

1.2 Design Principles

These are Netflix's published design principles for Hystrix

- Preventing any single dependency from using up all container (such as Tomcat) user threads.
- Shedding load and failing fast instead of queueing.
- Providing fallbacks wherever feasible to protect users from failure.
- Using isolation techniques (such as bulkhead, swimlane, and circuit breaker patterns) to limit the impact of any one dependency.
- Optimizing for time-to-discovery through near real-time metrics, monitoring, and alerting

1.3 Design Principles (continued)

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

1.4 Cascading Failures

- Failure of a single service causes failure of upstream or downstream services
- Hystrix uses the following to prevent cascading failures:
  - Bulkhead pattern

- ◊ Circuit Breaker pattern
- ◊ Thread Pooling

1.5 Bulkhead Pattern

Problem(s)

- Failure of a service affects all consumers of that service
- Failure of a service may affect a client's ability to connect to other services
- A client may make multiple request to a service, exhausting resources and causing other clients to fail

Solution(s)

- Partition services into groups for different clients
- Assign connection pools to clients for each service

Benefit(s)

- Isolate services (and clients to prevent cascading failures)
- System provides partial functionality in the event of service failure
- Different bulkheads may offer services with different QOS

Note

Bulkhead is a nautical term. In this case the pattern is named after the dividers in a ship that create watertight compartments so that a failure or leak in one compartment will not flood the entire ship.

1.6 Circuit Breaker Pattern

Problem(s)

- Clients may continue to attempt to access a service after the service has failed

- Clients may be unaware of a service failure until a request fails by timeout
- Clients accessing a failed server should be able to recover quickly and gracefully

Solution(s)

- Route client requests to servers with the highest probability of success
- Avoid recently failed servers
- Avoid servers under high load

Benefit(s)

- Fewer request get routed to failed services
- Failures may be detected earlier (not have to wait for back-end service)

Note

Circuit breaker is an electrical term. An electrical circuit breaker is *tripped* when the load on a circuit is too high. It is a defensive mechanism to prevent the circuit from physically overheating and causing a fire or worse.

1.7 Thread Pooling

- Rather than create new threads for every request
  - Threads are allocated from a pool
  - Prevents thread starvation
  - Prevents CPU over-utilization

1.8 Request Caching

- Reduces calls from Hystrix to backend service

    - Improves performance
- In memory cache
-

1.9 Request Collapsing

- Multiple requests are made for a *non-cached* service
- Hystrix only makes one request to the service
    - Other requests wait until the single request is made and cached
    - Reduces the overhead of multiple calls
    - Reduces load on target service

1.10 Fail-Fast

- Some requests should fail quickly and return
    - No retires
    - No fallback or alternatives
- A Circuit Breaker Open Circuit fails fast
- Example
    - Writes when a database is not available
    - Time-critical reads

1.11 Fallback

- Some requests can and should be retried in case of failure
- When a service is down requests can fallback to an in-memory cache

- Example
  - Read when a database is not available

1.12 Using Hystrix

- Ribbon and Zuul use Hystrix
- Applications can use Hystrix without using Ribbon or Zuul
- Hystrix provides a monitoring console
  - Works with Turbine and Eureka to identify and aggregate service data

1.13 Circuit Breaker Configuration

- **@HystrixCommand** annotation enables Hystrix Circuit Breaker
  - Synchronous methods
  - Asynchronous methods may return **Future<T>** or **Observable<T>**
- Annotation properties configure Short Circuit behavior
  - Failure threshold (%) to trip circuit,
  - Length of time to ignore open circuit
  - …

```
@HystrixCommand
@RequestMapping(value="cities/{zip}")
public String getCity() {
  String city = // query database for city by zip
  return city;
}
```

1.14 Fallback Configuration

- Fallback is related to Circuit Breakers
- When a Circuit Breaker is open the operation can either:
  - Fail Fast: return immediately
  - Fallback: attempt to satisfy request using another method
- Fallback is a property of the **@HystrixCommand**

```
@HystrixCommand(fallback="getCachedCity")
@RequestMapping(value="cities/{zip}")
public String getCity() {
   ...
}

// invoked as fallback if getCity() fails
@HystrixCommand
public String getCachedCity() {
   ...
}
```

Note

Spring will call **getCity()** for requests like this one: h**ttp://host:port/cities/90210**. If requests start failing and the Circuit Breaker trips open, then requests will be handled by **getCachedCity()**.

1.15 Collapser Configuration

- Define a handler method in the RestController
- Create a *collapser* service
  - Define a simple *dummy method* to be called by the handler method
  - Define a method to do aggregation

1.16 Rest Controller and Handler

```
@RestController
class OrderHistoryController {

  @Autowired
  private OrderHistoryCollapserService ohcService;

  @RequestMapping(value = "/order-history/{account}")
  public MessageWrapper getHistory(@PathVariable int account)
         throws ExecutionException, InterruptedException {
    return ochService.getHistory(account).get();
  }
}
```

Note

The handler method calls the *dummy* method of the service. On the next slide you will see that the *dummy* method returns a Future<MessageWrapper> object. The handler returns the result of the Future object's *get()* method.

1.17 Collapser Service (Part 1)

```
@Service
public class OrderHistoryCollapserService {

 // dummy method
 @HystrixCollapser(
   scope=com.netflix.hystrix.HystrixCollapser.Scope.GLOBAL,
   batchMethod="getHistories")
  public Future<MessageWrapper> getHistory(Integer account) {
    throw new RuntimeException("Should not be invoked");
  }

  // aggregator method
```

```
  @HystrixCommand
  public List<MessageWrapper> getHistories(
                                          List<Integer> accounts) {
    List<MessageWrapper> orderHistories =
                                 new ArrayList<>(accounts.size());

    // make calls to populate orderHistories

    return orderHistories;
  }
}
```

Note

The *dummy* method need the **@HystrixCollapser** annotation. It specifies that the batch or aggregation method id **getHistories()**. Notice that the dummy method throws an exception if it ever gets invoked.

The aggregator method, **getHistories** takes a List of accounts and requires the **@HistirxCommand** annotation.

1.18 How the Collapser Works

- Multiple RESTful GET requests to **/order-history/{account}**
- Hystrix queues up the calls
- When the time threshold is reached the queued account number are passed to the aggregator method
- The aggregator method then makes a single coarse grained request to fulfill *all* the individual RESTful requests
  - The coarse-grained request doesn't have to be a Web service it could be a SQL query
- To callers making RESTful requests the collapser is invisible

1.19 Hystrix Monitor

- Spring Boot web application
- **@EnableHystrixDashboard**
- Monitors circuits

```
@EnableHystrixDashboard
@SpringBootApplication
public class DashboardApplication {
  public static void main(String[] argv){
    SpringApplication.run(
              DashboardApplication.class,
              argv);
  }
}
```

1.20 Enable Monitoring

- Every Hystrix enabled app streams data

http://app-host/hystrix.stream

- Start

http://hystrix-host/hystrix/monitor?stream=http://app-host:port/hystrix.stream

- Using this mechanism Hystrix can only monitor one service
- In the next slide we will introduce Turbine

1.21 Turbine

- Turbine is an aggregator for Hystrix stream data
  - Discovers Hystrix services
  - Turbine streams aggregate data at http://turbine-host/hystrix.stream

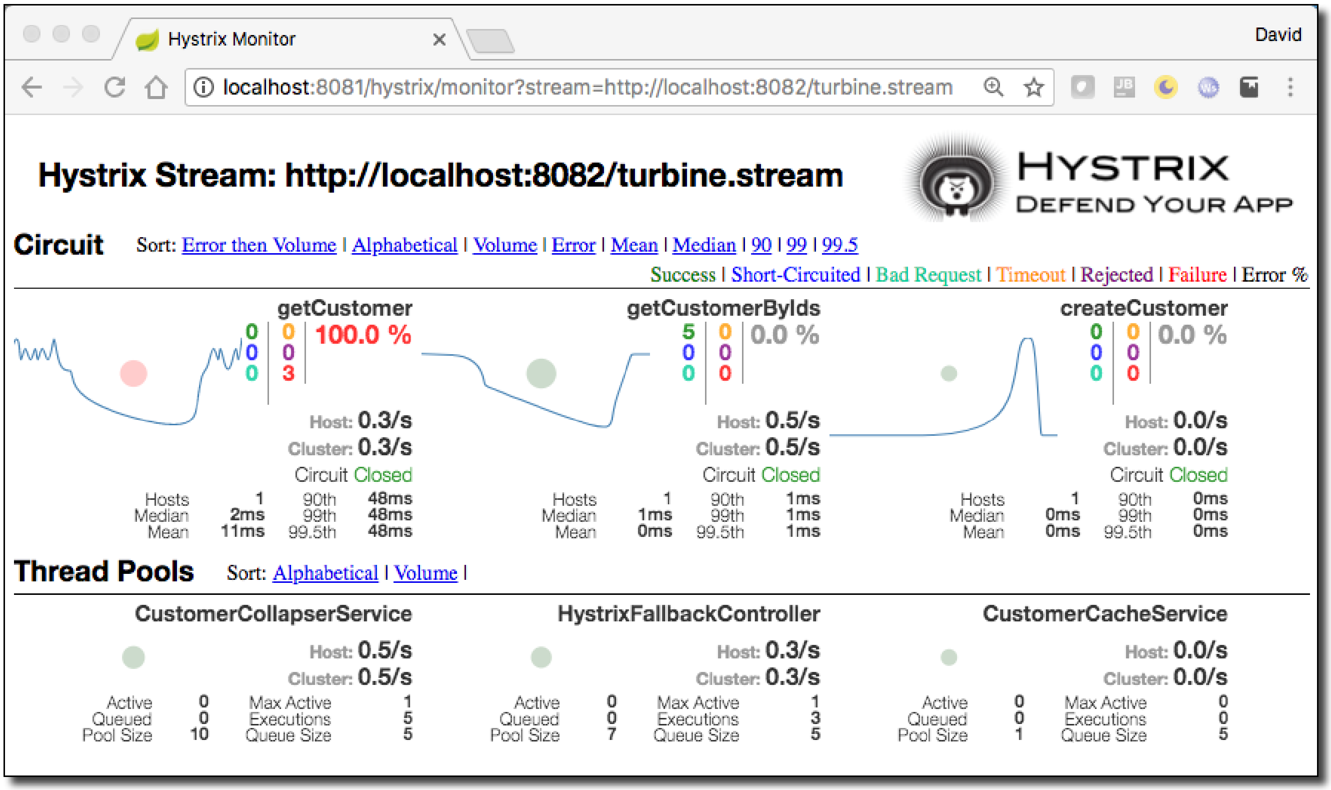
- Use the **@EnableTurbine** annotation

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

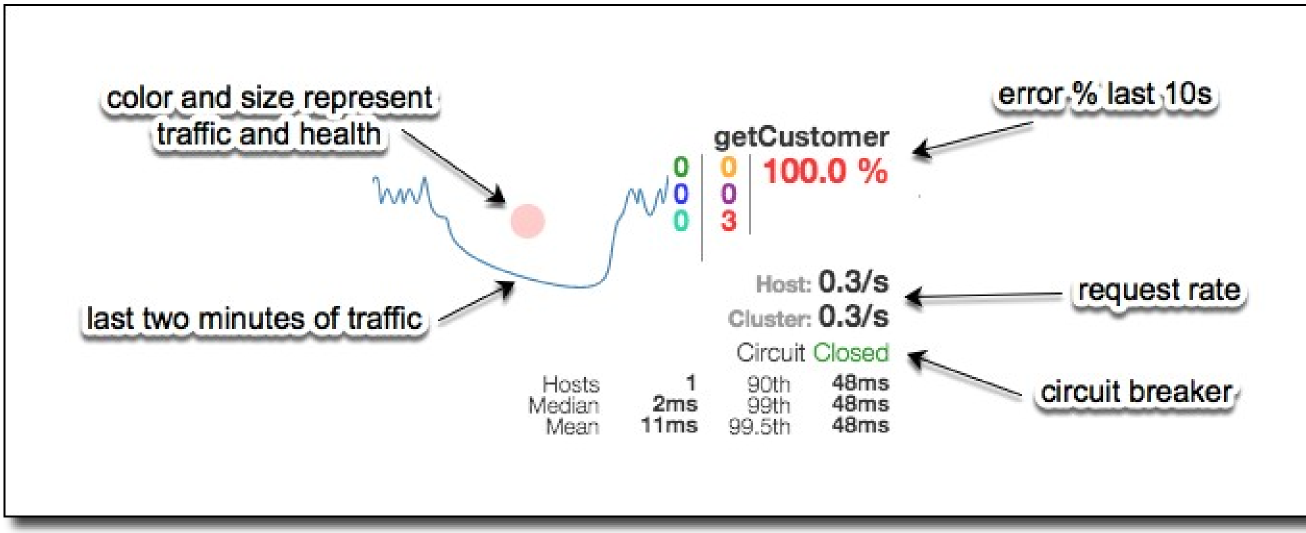
- Invoke the Hystrix monitor at:

http://hystrix-host/hystrix/monitor?stream=http://turbine-host/hystrix.stream

1.22 The Monitor



1.23 Monitor details



1.24 Summary

In this module we examined

- Netflix Hystrix
- Patterns
- Circuit Breaker Configuration
- Fallback Configuration
- Collapser Configuration
- The Monitor