Hystrix

G.O.A.T

What is Hystrix?

- A library that helps a developer control the interactions between the distributed microservices
- Done via isolation of points between the services, stopping cascading failures, and providing fallback options
- Increases a system's resiliency.

The Problem

A large, distributed system has a multitude of smaller, interlinked services

These services are prone to failure or delayed responses. If a service fails it may impact on other services affecting performance and possibly making other parts of application inaccessible or in the worst case bring down the whole application.

What is Hystrix used for?

- Gives protection from and control over latency and failures from accessed dependencies
- Stopping cascading failures across a distributed system
- Rapid recovery
- Enable almost real time monitoring
- Reduce downtime and allow the 4 9s

How does Hystrix work?

At a high level

- Wrapping all external systems or dependencies in a HystrixCommand or HystrixObservableCommand object that executes in another thread.
- Timing out calls that take longer than a user defined threshold.
- Maintaining a small thread pool for each dependency. If full requests to that dependency are rejected.
- Measuring successes, failures, timeouts, and thread rejection.
- Tripping a circuit breaker to stop all requests to a particular service for a period of time
- Performing fallback logic when a request ailes, is rejected, times out , or short circuits.

Construct a HystrixCommand or HystrixObservableCommand

- First step necessary to use Hystrix is to make either a HystrixCommand or an ObservableCommand object to represent the request being made to the dependency
- A HystrixCommand is used if a dependency is expected to return a single response
- A HystrixObservableCommand object is used if the dependency is going to return an observable that emits responses.
- Both HC and HOC should wrap code that will execute risky functionality. This is typically service calls over a network.

Command Execution

There are 4 ways to execute commands

- execute() - blocks then returns the single response received from the dependency
- queue() - returns a Future with which you can obtain the single response from the dependency
- observe() - subscribes to the observable that represents the responses from the dependency and returns an observable that replicates that source observable
- toObserveable() - returns an observable that when subscribed will execute the hystrix command and emit its responses

Execute and queue are only available to HystrixCommand objects

Circuits

- When the command is executed, Hystrix checks with the circuit breaker to see if the circuit is open or not.
- If the circuit is open, Hystrix will not execute the command but will route the flow to get the fallback
- If the circuit is closed the flow goes to check if there is capacity available to run the command.

run() and construct()

- HystrixCommand.run() returns a single response or throws an exception
- HystrixObservableCommand.construct()- returns an observable that emits the responses or sends an onError notification.

The Fallback

Fallback provides a means for graceful degradation. Hystrix tries to revert to your fallback when

- Command Execution fails
- When an exception is thrown by construct() or run()
- When command is short circuited because circuit is open
- When command's thread pool and queue are at capacity
- Command has exceeded timeout length

Fallback should be a generic response without network dependencies. EIther from in memory cache or some other static logic.

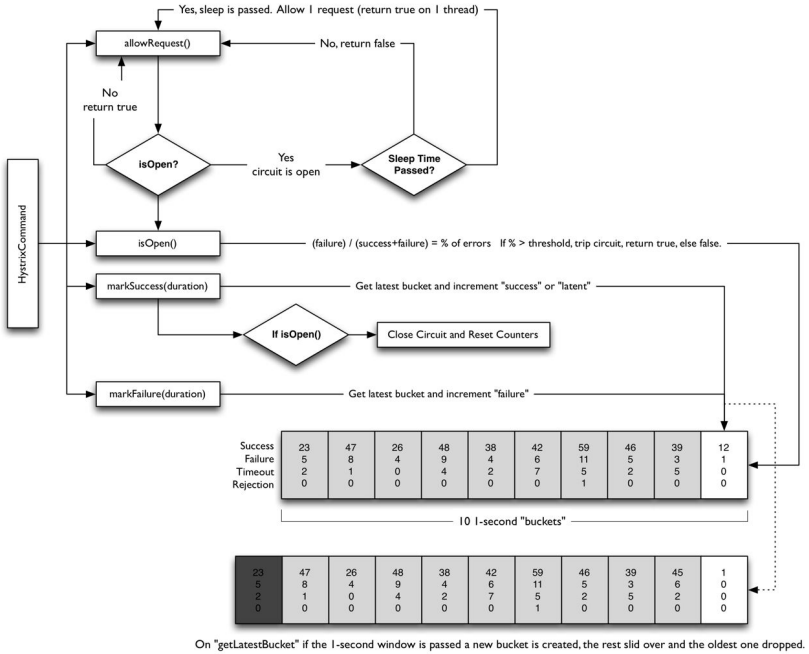
Fallback (cont).

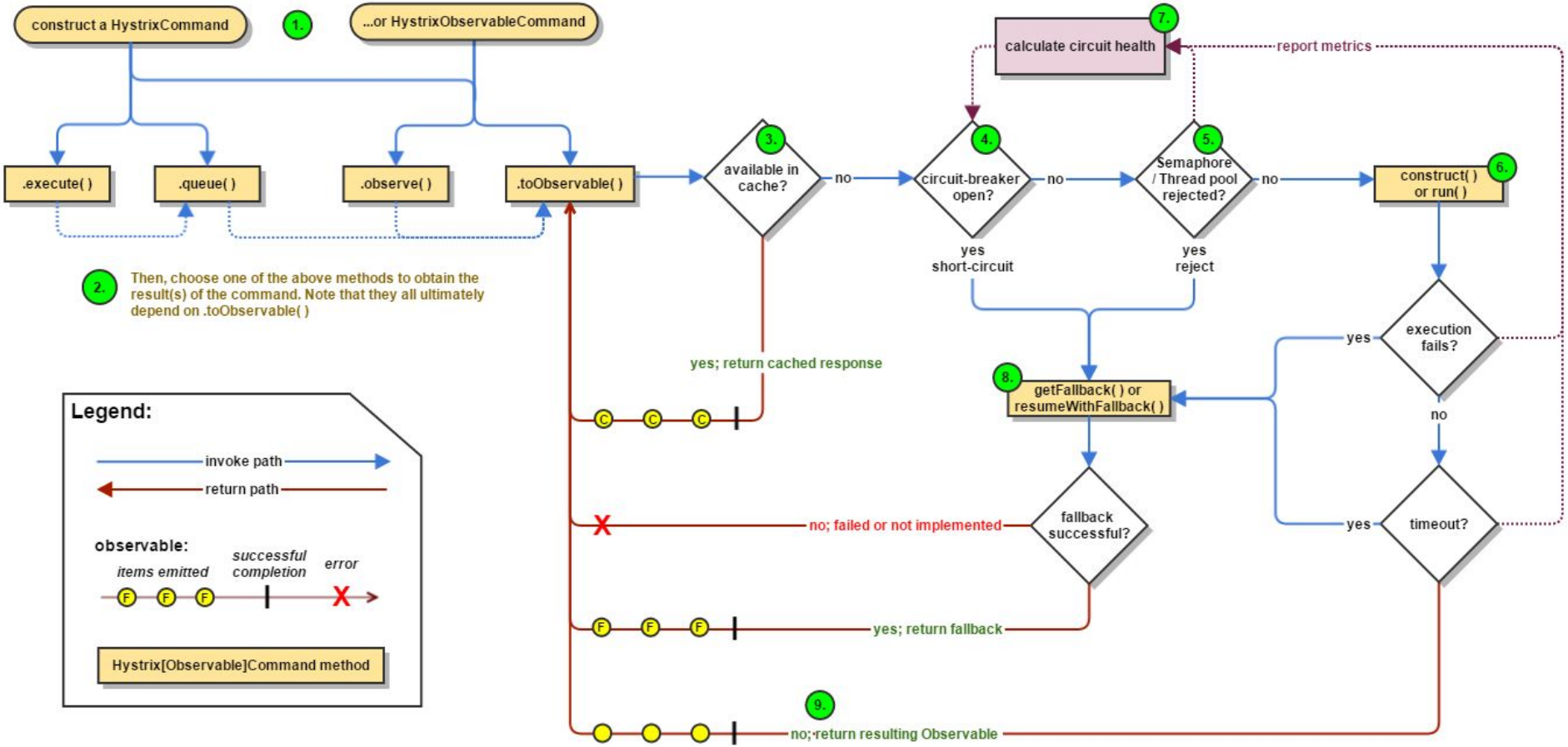
- Essentially provides a default value when something fails.

```
public class CommandHelloFailure extends HystrixCommand<String> {

    private final String name;

    public CommandHelloFailure(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        throw new RuntimeException("this command always fails");
    }

    @Override
    protected String getFallback() {
        return "Hello Failure " + name + "!";
    }
}
```

Interaction with the circuit breaker

- The HystrixCircuitBreaker is hooked into a HystrixCommand exeuction and will stop allowing execution if failures have gone past a certain amount.
- It will then allow single retries after a period of sleep (sleepWindow) until execution succeeds where it will close the circuit and allow execution again




construct a HystrixCommand
1.
...or HystrixObservableCommand
.execute( )
.queue( )
.observe( )
.toObservable( )
2.
Then, choose one of the above methods to obtain the result(s) of the command. Note that they all ultimately depend on .toObservable( )
3.
available in cache?
no
yes; return cached response
4.
circuit-breaker open?
no
yes
short-circuit
5.
Semaphore / Thread pool rejected?
no
yes
reject
6.
construct( ) or run( )
7.
calculate circuit health
report metrics
execution fails?
yes
no
timeout?
yes
8.
getFallback( ) or resumeWithFallback( )
fallback successful?
no; failed or not implemented
yes; return fallback
9.
no; return resulting Observable
Legend:
invoke path
return path
observable:
items emitted
successful completion
error
Hystrix[Observable]Command method

Sources

http://www.baeldung.com/introduction-to-hystrix

https://github.com/Netflix/hystrix/wiki

https://github.com/Netflix/Hystrix/wiki/How-To-Use

https://github.com/Netflix/Hystrix/wiki/How-it-Works