Hystrix Circuit Breaker with Spring BOOT

**Netflix Hystrix?**

In distributed systems or microservice based applications consist of many services communicate each other frequently at high scale. These services often prone to failed or faces problems like delayed response at high scale or due some other n/w or i/o related issues. In any application, if a service fails then it may lead to degrade in other service’ performance and sometimes making the entire application down even.

However, we have many solutions to above problem. One of such is Hystrix framework from Netflix which make the application reliable & fault tolerant.

For the sake of understanding the fault tolerant behavior of application, we may consider the example of electronic circuit in any home or other. If short circuit takes place in any one of the connection the circuit breaker trips and make itself open (like in MCB & RCB) which make all the connection to that circuit failed.

**Hystrix in Spring?**

Spring cloud provides building web application of high performance. It may incorporate the REST API for communication in case of microservice architecture. Spring provides *Spring Cloud Netflix Hystrix* – a fault tolerant library which takes care of it.

Spring supports Hystrix annotations to be configured in application. With spring boot application, the configuration is always ease with *application.properties* file, which contains common Hystrix properties along with others.

These are following component to provide basic idea of Spring Batch.

Click with ctrl to navigate to these topics.

1. [Setting up Hystrix in spring boot](#setup1)
2. [Apply fault tolerance in spring application](#applyfault2)
3. [Hystrix Command Properties - Execution](#executionproperties3)
4. [Hystrix Command Properties - Fallback](#fallbackproperties4)
5. [Hystrix Command Properties - Circuit Breaker](#circuitproperties5)
6. [Hystrix Thread Pool Properties](#threadproperties6)
7. **Setting up Hystrix in Spring Boot Application**

To use hystrix framework in spring requires following dependency:

compile group: 'org.springframework.cloud', name: 'spring-cloud-starter-hystrix', version: '1.3.4.RELEASE'

This will add Hystrix dependency in application.

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

1. **Apply Fault Tolerance in Spring Application**

Hystrix framework provide annotation to make fault tolerant enable.

@EnableCircuitBreaker - Annotation to enable a CircuitBreaker implementation.

@HystrixCommand - Annotation used to specify some methods which should be processes as hystrix commands.

The first annotation will enable circuit breaker in application. This annotation can be put at any configuration bean. Without this hystrix won’t work. Second annotation @HystrixCommand is method level annotation, responsible to apply circuit breaker for that method. To facilitates fault tolerance for particular methods, annotate those methods with @HystrixCommand.

@HystrixCommand has following useful attributes:

**fallbackMethod** - Specifies a method to process fallback logic. A fallback method should be defined in the same class where is HystrixCommand. The fallback method should have same signature to a method which was invoked as hystrix command.

To test this property of hystrix command, start hystrix-consumer application. Now, can make rest call using below uri, this will result in exception because hystrix-producer is not up & running. Hence, fallback logic will be executed as per the configuration.

Rest uri: <http://localhost:8077/case1/show/zeta>

Expected Output: Hello user ! [ As we are passing zeta as value even though, getting default value – user.

For example:

@Service("fallback")

**public** **class** MessageServiceImpl1 **implements** MessageService {

**private** **static** Logger *logger* = LoggerFactory.*getLogger*(MessageReceiverController.**class**);

/\*\*

\* This is simple implementation of hystrix with only fallback logic.

\*/

@HystrixCommand(fallbackMethod = "failedCall")

**public** String showMessage(String username) {

String result = **new** RestTemplate().getForObject("http://localhost:8088/get/"+username, String.**class**);

**return** result;

}

/\*\*

\* This will be invoked by hystrix circuit breaker if call failed to annotated method

\*

\* **@param** username

\* **@return** default message upon failure

\*/

**public** String failedCall(String username, Throwable t) {

*logger*.error("Fall Back called for exception: " + t.getMessage());

**return** "Hello " + "user !";

}

}

**commandProperties** - Specifies HystrixCommand properties. We shall see properties in coming sections.

**commandKey** - Specifies HystrixCommand key. It is used to configure instance properties. Hystrix supports global configuration of all methods and also specific method configuration also using instance property.

For example, to control hystrix properties for particular methods, use command key.

@HystrixCommand( commandKey = “showKey”, fallbackMethod = “failedCall” )

**public** String showMessage(String username) {

String result = **new** RestTemplate().getForObject("http://localhost:8088/get/"+username, String.**class**);

**return** result;

}

Now, in application.properties, we can give specific property for all the annotated methods having command key as = “showKey”.

#specific instance property

hystrix.command.showKey.execution.isolation.thread.timeoutInMilliseconds=15000

**ignoreExceptions** - Defines exceptions which should be ignored. Normally, any exception will cause the fallback method to be invoked upon failure of the annotated methods. Giving this property will ignore that exception and will not trigger fall back call. Normal exception will cause the failure of method.

For example, we are generating a runtime exception based on given input of the call. Here, if input is demo, then it will cause an exception. Normally, it should invoke the fallback logic, but as we have specified ignore exceptions property, it will not invoke fallback and will result in normal exception flow:

@Service("ignoreException")

**public** **class** MessageServiceImpl3 **implements** MessageService {

**private** **static** Logger *logger* = LoggerFactory.*getLogger*(MessageReceiverController.**class**);

@HystrixCommand( fallbackMethod="failedCall",

ignoreExceptions = CustomException.**class** )

**public** String showMessage(String username) {

**if** ("demo".equals(username.trim().toLowerCase()))

**throw** **new** CustomException ("dummy name passed");

String result = **new** RestTemplate().getForObject("http://localhost:8088/get/"+username, String.**class**);

**return** result;

}

**public** String failedCall(String username, Throwable t) {

*logger*.error("Fall Back called for exception: " + t.getMessage());

**return** "Hello " + "user !";

}

}

To test this feature, make a rest call with given Uri: [http://localhost:8077/case3/show/{username}](http://localhost:8077/case3/show/%7busername%7d)

<http://localhost:8077/case3/show/demo>

Now making this call will result in proper execution if hystrix-producer app is running, but it doesn’t then will result into fallback output. But if make request with username as demo which will cause in raising custom exception as below implementation and will show the normal exception regardless of hystrix fallback configuration.

**public** **class** CustomException **extends** RuntimeException {

**public** CustomException(String message, Throwable cause) {

**super**(message, cause);

}

**public** CustomException(String message) {

**super**(message);

}

}

1. **Hystrix Command Properties - Execution**

The Execution properties control how method will be execute with HystrixCommand.

Execution properties:

- As far we have seen fallback logic execution upon failure of the any service method. Hystrix also given features to control the execution of method. Suppose we have situation like any method taking much more time for execution which makes other resources block of engaged for that time. Or sometimes, a method may have stuck while execution and may never be completed. Well in these cases, it is quite useful that we control execution time of the method

For this purpose, hystrix comes with two execution properties as

**execution.timeout.enabled & execution.isolation.thread.timeoutInMilliseconds**

**execution.timeout.enabled**

- This property will enable the execution timeout for the method execution. Default behavior of this property is enabled which means we can give specific timeout for specific methods to get its task done.

|  |  |
| --- | --- |
| **Default Value** | true |
| **Default Property** | hystrix.command.default.execution.timeout.enabled |
| **Instance Property** | hystrix.command.HystrixCommandKey.execution.timeout.enabled |

**execution.isolation.thread.timeoutInMilliseconds**

- As stated above, this property will ensure that method annotated will execution within the given time. If not then it will stop the execution of method and will result in fallback logic execution. We can configure this behavior globally in properties file and at method level also in case we need specific time for some methods. Note that this property will depend on previous property which will enable this behavior. For example, consider the below code which will make the method to be carried within 5 seconds only, after that timeout exception will be thrown.

@Service("timeoutInterrupt")

**public** **class** MessageServiceImpl4 **implements** MessageService {

**private** **static** Logger *logger* = LoggerFactory.*getLogger*(MessageReceiverController.**class**);

@HystrixCommand(fallbackMethod = "failedCall", commandProperties={@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")})

**public** String showMessage(String username) {

**try** {

Thread.*sleep*(6000);

} **catch**(InterruptedException ie) {

*logger*.error("Exception : " + ie);

}

String result = **new** RestTemplate().getForObject("http://localhost:8088/get/"+username, String.**class**);

**return** result;

}

**public** String failedCall(String username, Throwable t) {

*logger*.error("Fall Back called for exception: " + t);

**return** "Hello " + "user !";

}

}

To check this property with sample application, make rest call to following Uri: <http://localhost:8077/case4/show/demo>

As stated in code, minimum time required for carrying this method execution will be not less 6 seconds, but we have given timeout as 5 seconds resulting in fallback logic execution.

Exception thrown is: ***com.netflix.hystrix.exception.HystrixTimeoutException***

|  |  |
| --- | --- |
| **Default Value** | 1000 |
| **Default Property** | hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds |
| **Instance Property** | hystrix.command.*HystrixCommandKey*.execution.isolation.thread.timeoutInMilliseconds |
| **Annotation Based** | @HystrixCommand(fallbackMethod = "failedCall", commandProperties={@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")}) |

**execution.isolation.thread.interruptOnTimeout**

- Using this property will controls the flow of execution between method and fallback. For instance, we can assume the previous scenario. As the timeout is 5 seconds and method will take more than 6 seconds resulting in interruption of execution.

If interruptedTimeout is true which is the default behavior, then method will be executed normally even after timeout but will be followed by fallback method. Hence, we will get fallback logic output only.

But, if value is set to be false, then soon after getting the timeout exception, fallback logic is immediately executed and result is published also and then method execution takes place. Either the case, we will get fallback logic output.

|  |  |
| --- | --- |
| **Default Value** | true |
| **Default Property** | hystrix.command.default.execution.isolation.thread.interruptOnTimeout |
| **Instance Property** | hystrix.command.*HystrixCommandKey*.execution.isolation.thread.interruptOnTimeout |

1. **Hystrix Command Properties - Fallback**

It controls how fallback logic is executed upon method failures.

Fallback properties:

**fallback.enabled**

- By default, this property is enabled, which means after failure of the method execution, the given fallback logic will be executed. If it is enabled and fallback method is not given then will result in normal flow based on exception causing the failure.

If fallback method is given in hystrix using annotation, and this property is disabled by giving false value to it, then fallback method execution will take place, again resulting in normal flow of execution.

|  |  |
| --- | --- |
| **Default Value** | true |
| **Default Property** | hystrix.command.default.fallback.enabled |
| **Instance Property** | hystrix.command.HystrixCommandKey.fallback.enabled |

**fallback.isolation.semaphore.maxConcurrentRequests**

- This property sets the maximum number of requests allowed to make from the calling thread. It controls the number of concurrent fallback execution. If the maximum concurrent limit is hit then subsequent requests will be rejected and an exception thrown since no fallback could be retrieved.

For example, if value is given to be 2 then we few concurrent calls are made for method execution and any two of calls failed in successful execution. Then not more than two threads or requests are allowed to execute the fallback logic. If any third request failed and tries to execute the fallback logic concurrently with other two, then it will result in exception which is: ***com.netflix.hystrix.exception.HystrixRuntimeException: showMessage fallback execution rejected***.

|  |  |
| --- | --- |
| **Default Value** | 10 |
| **Default Property** | hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests |
| **Instance Property** | hystrix.command.HystrixCommandKey.fallback.isolation.semaphore.maxConcurrentRequests |

To test this property, start the hystrix-consumer, at make few rest calls to this Uri: <http://localhost:8077/case3/show/zeta>

As producer app is not started, it will result in failure of method and will tries to fallback logic. Now, make break point in fallback logic to stop current thread from completing the execution. As the third thread will try to execute the fallback will get exception.

1. **Hystrix Command Properties – Circuit Breaker**

A screenshot of a cell phone

Description generated with very high confidenceIt's common for distributed applications to make remote calls to software running in different processes or different service in microservice based applications, probably on different machines across a network. One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response or sometimes just may stuck its execution by holding some costly resource.

It may result into failure of application due to some service fail due to n/w congestion or connection problem or any other. Circuit breaker is a design pattern to handle such problems. Hystrix comes with circuit breaker features which can be configured efficiently with below given properties.

The circuit breaker properties control the behavior of the circuit upon request failure or rejection. Using this property, circuit tripping, opening, closing can be controlled.

As we can see in picture, client making few requests to access the service. But due to some internal difficulties of service provider, it may result into failure. Based on default configuration or specific one, after a fixed attempts failure, circuit will trip and making it open. Now, all the request for given time interval will not land to service implementation. This will eventually allow few seconds to service to recover from the failure.

**circuitBreaker.enabled**

- As stated above, this will enable the circuit breaker design patter in hystrix annotated methods. By default, it is enabled, so we can disable also. It also decides other similar property to be applied or not. Such as **requestVolumeThreshold, sleepWindowInMilliseconds, etc..** property will work if it is enabled.

This property determines whether a circuit breaker will be used to track health and to short-circuit requests if it trips. By default, it is enabled which will trip and block other subsequent request from executing the method logic based on another configuration.

|  |  |
| --- | --- |
| **Default Value** | true |
| **Default Property** | hystrix.command.default.circuitBreaker.enabled |
| **Instance Property** | hystrix.command.HystrixCommandKey.circuitBreaker.enabled |

**circuitBreaker.requestVolumeThreshold**

- This is to specify the number of requests failure within rolling window time (10s) which will trip the circuit to open and all subsequent call will not go to actual method implementation.

This property sets the minimum number of failed requests which will trip the circuit. For example, suppose a threshold limit is 5, then if total of 5 requests among all is failed also. It will not trip the circuit. If more than 5 requests failed then circuit will be tripped which means actual method logic will not be executed and fallback logic will be executed directly. It will help to recover the resource.

|  |  |
| --- | --- |
| **Default Value** | 20 |
| **Default Property** | hystrix.command.default.circuitBreaker.requestVolumeThreshold |
| **Instance Property** | hystrix.command.HystrixCommandKey.circuitBreaker.requestVolumeThreshold |

**circuitBreaker.errorThresholdPercentage**

It is like previous property. This property is used to specify the percentage of failed requests within rolling window (10s). Assume error percentage is 50 then, within rolling window of 10s, if 4 requests are made of which 2 are failed. Then it will trip the circuit, and next calls will be not allowed to execute called method, instead it will go to fall back logic.

|  |  |
| --- | --- |
| **Default Value** | 50 |
| **Default Property** | hystrix.command.default.circuitBreaker.errorThresholdPercentage |
| **Instance Property** | hystrix.command.HystrixCommandKey.circuitBreaker.errorThresholdPercentage |

**circuitBreaker.sleepWindowInMilliseconds**

- If circuit trips/opens then all calls are not served by actual implementation. This behavior will continue for some time. This particular property will specify the time interval for skipping the requests. After that, all request will try to execute actual method call.

As discussed in the previous property, which controls no of failed request to trip the circuit. After tripping the circuit, all request will go to fallback. This property sets the amount of time the method will not be executed and circuit will be closed. After this amount of time, requested method will be attempted for execution.

|  |  |
| --- | --- |
| **Default Value** | 5000 |
| **Default Property** | hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds |
| **Instance Property** | hystrix.command.HystrixCommandKey.circuitBreaker.sleepWindowInMilliseconds |

To test these properties, add below properties in application.properties as:

#circuit breaker properties

hystrix.command.default.circuitBreaker.enabled=true

hystrix.command.default.circuitBreaker.requestVolumeThreshold=2

hystrix.command.default.circuitBreaker.errorThresholdPercentage=50

hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=15000

For our sample example, start hystrix-consumer sample application, don’t start producer as we want failure calls. Now, make a few rest call to this Uri: <http://localhost:8077/case5/show/dem>

As in sample properties, make few breakpoint in showMessage() & failedCall() of MessageServiceImpl5 of consumer application to check. Now, after two unsuccessful execution of showMessage(), circuit will trip. We can observe by making subsequent calls again & again that calls are not going inside showMessage(), instead directly lands to fallback method only for 15 seconds as given in property.

@Service("circuitBreaker")

**public** **class** MessageServiceImpl5 **implements** MessageService {

**private** **static** Logger *logger* = LoggerFactory.*getLogger*(MessageReceiverController.**class**);

@HystrixCommand(fallbackMethod = "failedCall", commandProperties={@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")})

**public** String showMessage(String username) {

String result = **new** RestTemplate().getForObject("http://localhost:8088/get/"+username, String.**class**);

**return** result;

}

**public** String failedCall(String username, Throwable t) {

*logger*.error("Fall Back called for exception: " + t.getMessage());

**return** "Hello " + "user !";

}

}

**circuitBreaker.forceOpen**

- This property forcefully open/trip the circuit. If it is true then all the coming requests will not executed the actual method, instead will lands to fallback method.

|  |  |
| --- | --- |
| **Default Value** | false |
| **Default Property** | hystrix.command.default.circuitBreaker.forceOpen |
| **Instance Property** | hystrix.command.HystrixCommandKey.circuitBreaker.forceOpen |

**circuitBreaker.forceClosed**

- Similarly, like forceOpened, if it is true will close the circuit, and request can be processed by actual method implementation.

|  |  |
| --- | --- |
| **Default Value** | false |
| **Default Property** | hystrix.command.default.circuitBreaker.forceClosed |
| **Instance Property** | hystrix.command.HystrixCommandKey.circuitBreaker.forceClosed |

1. **Hystrix Thread Pool Properties**

The following properties control the behavior of the thread-pools that Hystrix Commands execute on. Hystrix uses ThreadPoolExecutor to carry out thread based execution of the hystrix command. Default thread pool size is 10.

Thread Pool properties:

**coreSize**

- This property sets the core size of the thread pool managed by hystrix. Depending on application, we can set the pool size. However, if we don’t set it default size 10 will be used.

|  |  |
| --- | --- |
| **Default Value** | 10 |
| **Default Property** | hystrix.threadpool.default.coreSize |
| **Instance Property** | hystrix.threadpool.HystrixThreadPoolKey.coreSize |

For example, if the core size is given to be 2 then only two threads will be fired to carry out method execution. Now suppose, there is more than two concurrent requests and all two threads still processing the request. If any other request is made before completion of previous of any one then it will cause an exception with following stack trace:

Task java.util.concurrent.FutureTask@473ce29f rejected from [java.util.concurrent.ThreadPoolExecutor@6e1ec6e5](mailto:java.util.concurrent.ThreadPoolExecutor@6e1ec6e5) [Running, pool size = 2, active threads = 2, queued tasks = 0, completed tasks = 0]

Note: - These all properties, we’ve come across can be easily set by using application.properties file of spring boot application. Or else, can be set through @HystrixCommand annotation as we discussed in early section.

Let’s see the entire sample configuration for above all properties:

spring.application.name=hystrix-consumer

# spring hystrix properties

#execution based properties

hystrix.command.default.execution.timeout.enabled = true

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=15000

hystrix.command.default.execution.isolation.thread.interruptOnTimeout=false

#fallback properties

hystrix.command.default.fallback.enabled=true

hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests=5

#circuit breaker properties

hystrix.command.default.circuitBreaker.enabled=true

hystrix.command.default.circuitBreaker.requestVolumeThreshold=2

hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=15000

hystrix.command.default.circuitBreaker.forceOpen=false

hystrix.command.default.circuitBreaker.forceClosed=false

#threadpool properties

hystrix.threadpool.default.coreSize=2

#specific instance property

hystrix.command.customkey.execution.isolation.thread.timeoutInMilliseconds=15000

Read More properties: <https://github.com/Netflix/Hystrix/wiki/Configuration>

For the sample application, run both the applications [Hystrix-Producer, Hystrix-Consumer], now send the request to consumer using following uri. To check the fallback, close the producer application, again send the same request, it will execute fallback logic. To check other properties, send multiple request accordingly with the configured properties above.

<http://localhost:8077/show/zeta>