# Lab : Adding Resilience Using Hystrix

In this episode, w e w ill learn how to deal w ith netw ork failures, w hich are inevitable in a distributed environment. In order to do that, w e w ill introduce the circuit breaker architectural pattern and cover w hen it should be used, and w hat its benefits are. We w ill look at its Netflix implementation, Hystrix. We w ill also cover how it is implemented and how it can be used. As an example, w e w ill use Hystrix to add resilience and fault tolerance to our sample application.

### Pre-reqs:

- Openshift Wildfly Lab

## Unreliable network

When you develop your services for distributed environments, you must take into consideration that invocations of the services w ill be performed over the netw ork. As a result, the application must be prepared to deal w ith netw ork failures, w hich w ill certainly happen.

This problem is further enhanced by the fact that a single misbehaving service can poison a large number of services. Let's take a look at the number of scenarios, that can make this possible.

### Dependent services

In large distributed systems, each service w ould have a large number of dependencies on other services. It requires only one dependency to fail to make the service irresponsible. Also, a service w ill go dow n too, becoming unavailable for other services that depend on it. Such a situation is know n as cascading failures (Further reading, link 1).

How ever, that's not all. In a production-ready environment w hen there are a lot of calls being performed, a service that has latency problems w ill quickly block all available threads and make all other services unreachable.

Clearly, if w e w ant to design a robust distributed system, w e w ill need to have a tool that w ill enable users to deal w ith the problems described previously. The tool that w e w ill use is the Hystrix library.

Hystrix is a library developed by Netflix to deal w ith service failures and to provide robustness for its sophisticated distributed architecture. Let's find out how Hystrix deals w ith the problems previously described.

## The circuit breaker pattern

The architectural design pattern created to deal w ith the problems described previously is the circuit breaker pattern. The main idea behind it is simple: w rap the invocation code into the command, w hich w ill perform the invocation, and calculate the status of the remote service. If the service is declared unreachable by the metrics used by the command, then the next invocations are rejected immediately. After a given time, new connections w ill be tried again, and, if successful, the command w ill start performing invocations to the service again.

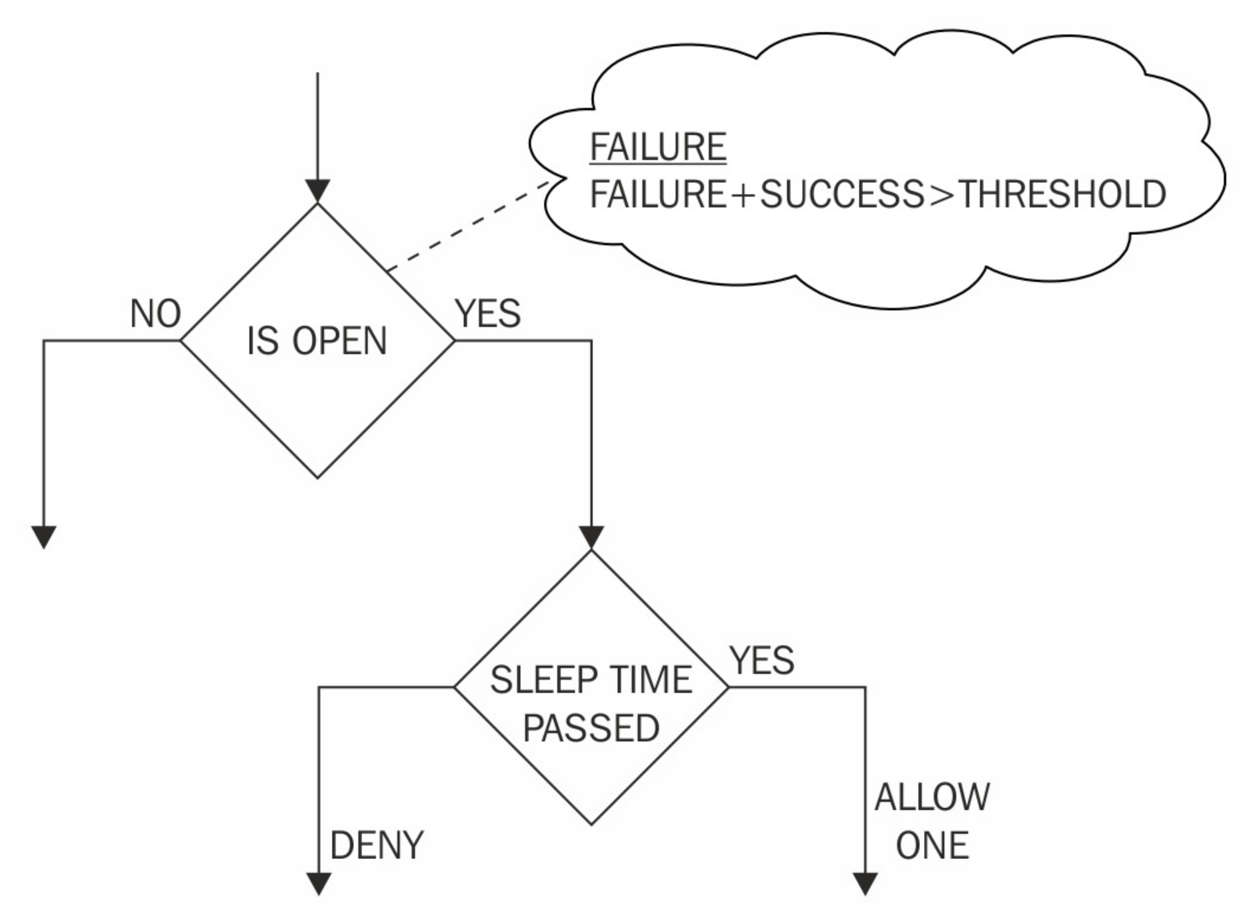
The name of the pattern w as taken from the electrical circuit breaker, a device used to protect the electrical circuit from the damage that may result from the excess electrical current. If the current in the circuit is too high then the circuit breaker opens, preventing the current from flow ing. In order to make the circuit operate again, the circuit breaker has to be closed.

Ow ing to its archetype, the softw are circuit breaker has inherited the electrical nomenclature. If the targeted service is healthy and the invocations are forw arded to it directly, w e w ill be talking about closed breaker. If the health metrics are exceed, the invocations are not performed, and the circuit breaker w ill be opened.

Obviously, the library that is responsible for the implementation of the circuit breaker has to provide algorithms that decide w hether the remote service is healthy, how and w hen to open the circuit, and w hat to do if a circuit breaker is closed. Let's discuss how Hystrix does it.

## The Hystrix circuit breaker

The follow ing diagram presents the behavior of the Hystrix circuit breaker:



During the invocation of the remote service, Hystrix checks w hether the circuit breaker is open. This decision is made based on the statistics gathered from the recent invocations. If the percentage of failures in the last time w indow is low er than the configured threshold, then the circuit is open and the invocation is performed.

After the invocation is performed, the circuit breaker stores its result (success/failure) in the statistics. The statistics are gathered for the configured time w indow, w hich is further divided into the number of buckets, w ith only one bucket being discarded at a time so that the data for the given w indow is not discarded all at once.

What happens w hen the circuit is open? Firstly, the algorithm checks w hether the configured sleep time has passed. If it is the case, then only one request is allow ed to be executed. This stage of a circuit breaker is called half-open and its purpose is to check w hether the invoked service is healthy again. If the invocation succeeds, then a circuit breaker is opened again and the metrics are reset. If, on the other hand, the sleep time has not exceeded or the one invocation in a half-open state has failed, then the circuit breaker is opened again and the sleep time is reset.

So, w e now know the Hystrix circuit breaker algorithm and how it reacts to statistics on successful and failed invocation. How ever, how do w e actually define failure? There are three cases w hen the invocation is marked as failed. First, the configured invocation timeout has been exceeded. Second, the client library has throw n an exception. Third, the number of threads available for a given dependency has been exceeded.

The last point is an implementation of the bulkheading algorithm. Let's learn more about it.

## Bulkheading

In order to prevent a situation in w hich one of the dependencies uses the w hole thread pool of an application, Hystrix keeps the thread pool for each dependency. If one of the dependencies becomes latent, it w ill keep all its threads busy and w ill reject further invocations, resulting in an increased failure count.Such a strategy is called **bulkheading**.

This time, the nomenclature is taken from ship engineering: the hull of the ship is divided into isolated bulkheads so that the hull damage in one place results in only one bulkhead being filled w ith w ater. Similarly, providing a thread pool for each of the dependencies results in only a dedicated thread pool being used if one of the services is misbehaving.

In complex distributed environments, it is often the case that the application has many dependencies, each of w hich depends on other client libraries. Often, such libraries are black boxes provided by a third-party company, making them hard to debug. In addition, increasing the number of those libraries increases the risk that one of them w ill *poison* the w hole application. With bulkheading, you can easily mitigate this risk.

## Note

The state of each client can be easily tracked by the status of its thread pool. If monitoring shows that one of the thread pools is full, it is an indicator that it should be examined. If the underlying problem is fixed, the thread pool will clear up and the service will continue its operation.

The dependencies that share the same thread pool are configurable. As a result, you are able to tune bulkheading behavior according to your architecture. Such a configuration is done using Hystrix group mechanism, which we will show you in the examples later in this episode.

So, we already know that the invocation may fail or be forced to fail by Hystrix. But, what happens in that scenario? The mechanism that is supposed to deal with invocation failures is called **fallbacks**. Let's learn more about it now.

## Fallbacks

Hystrix implements a fallbacks mechanism, which allows you to execute your code whenever the failure of the invocation happens. The command allows you to implement a fallback method, which will be executed during the failure. The method is executed regardless of the cause of the failure—the same method will be executed in case of timeout or thread pool overflow.

The fallback method doesn't have to be implemented. If fallback is not implemented, the exception thrown by Hystrix will be propagated down the stack trace.

If, on the other hand, you decide to implement the fallback, you have a number of strategies for doing that. Let's take a look at a few examples.

If you are using a service that is used to read a data, you can return an empty answer in case of invocation failure. In such a scenario, no data will be available in case of service failure. This solution hides the underlying failure and immediately returns the response. The problem is, obviously, that the requested data is unavailable. You can deal with that by implementing the local cache and return the latest response in case of failure. In this scenario, the failure will be hidden and the data will be available. It won't be up to date for the time of failure, but it will allow your system to continue its operation.

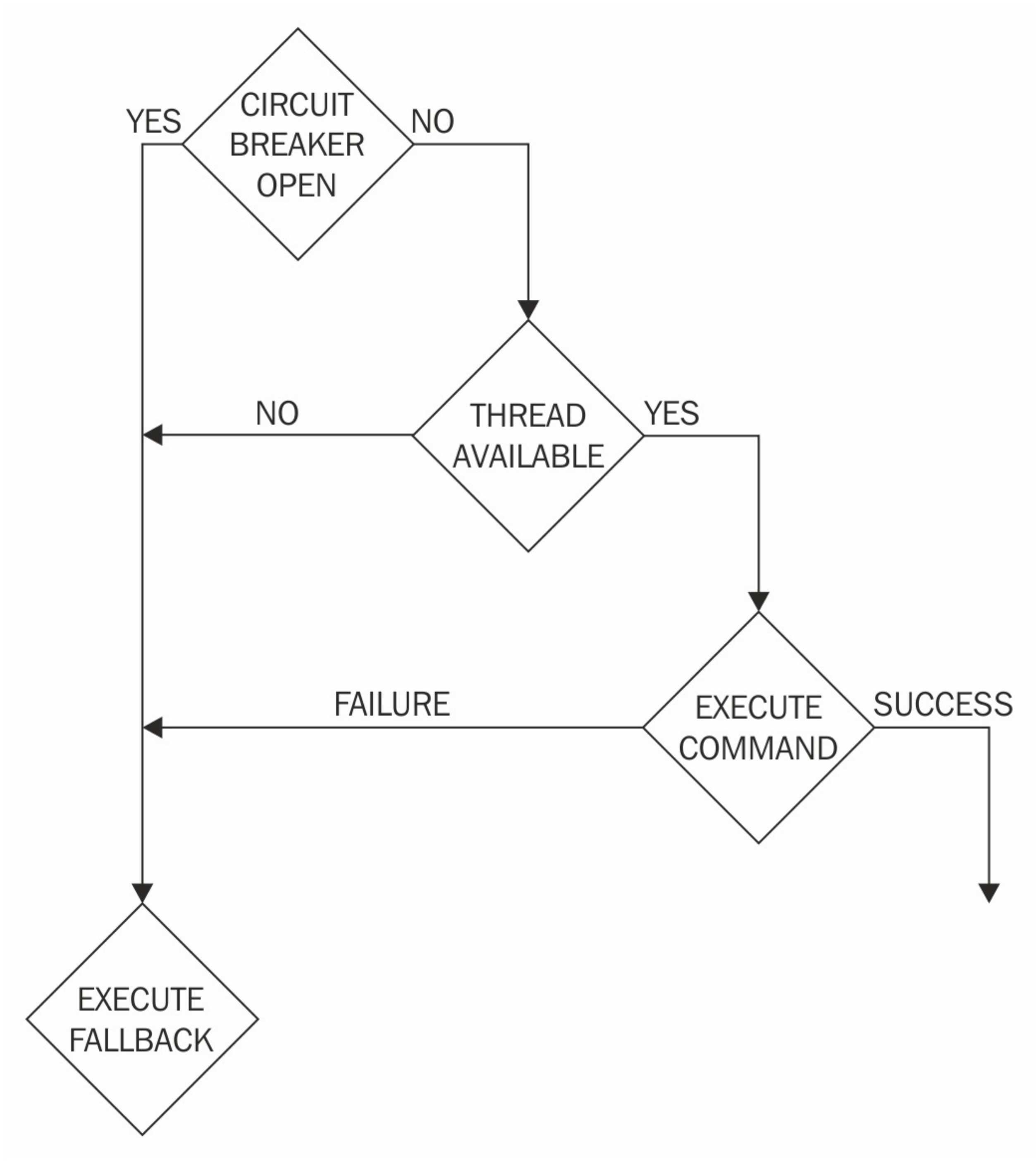
Let's suppose now that you are using the authorization service to decide whether the user is authorized to perform some further operations. In this case, you can implement the fallback, which will always return the same response. However, what should this response be? As usual, it depends on your use case. In some scenarios, you may want to avoid a situation when a user who has paid for a service is unable to use it. In this scenario, you would return the successful authorization each time. The drawback is that a number of users will be able to use content that they haven't paid for at the time of authorization service failure. In other scenarios, you may need to deny authorization for all users. Surely the temporary allow-all strategy is not suitable for the bank application. In this case, you would have to deny the authorization for all users.

Finally, in some scenarios, not writing the fallback is a good strategy. Let's suppose that you are implementing an invocation that is supposed to modify some data as a part of transactional operations. In this scenario, the propagated exception is the strategy that we want: the whole operation will be stopped, and the exception will be propagated to the transaction manager, which will roll back the transaction.

In this section, we have only hinted at a number of possible fallback implementation strategies. As you may have noticed, the specific implementation (or lack of) depends directly on your service's business requirements. The key point to remember is that Hystrix won't allow the network failure to compromise the behavior of your application, and if a failure occurs it will allow you to deal with it using the fallback mechanism.

## The whole algorithm

Finally, we are ready to sum up the behavior of the Hystrix library:



At the beginning, the user constructs the command and starts its execution. Hystrix checks w hether the circuit breaker associated w ith this command is closed. If the circuit breaker is open, then the invocation w ill be rejected immediately and the fallback w ill be executed (if implemented). If the circuit breaker is closed, then the thread-pool is checked. If there are no available threads in the thread-pool, then the invocation fails; optionally, the fallback is executed and the failure is reported to the circuit breaker. If, on the other hand, there are threads available, then the invocation starts. If the invocation misses its timeout, then the failure is reported to the circuit breaker and the optional fallback is executed.

## Note

In this scenario, the thread may be blocked. Hystrix will time it out, but will have to wait for the client library to *return the thread*. If the invocation finishes and fails, then the failure is reported to the circuit breaker and optionally the fallback is executed.

Finally, if the execution succeeds, then the success is reported to the circuit breaker and the response is returned as the result of the command execution.

You have already learned the basics of the Hystrix's circuit breaker implementation. Now it is time to learn its basic API, which we will use later in this episode. Let's do it now.

## Using Hystrix

---

In order to learn Hystrix behavior in practice, we are going to extend customer-gateway service so that it uses Hystrix for its invocations. Later, we are going to make one of our services artificially unresponsive and see how Hystrix behaves. Let's start.

### Note

Examples reference: `chapter11/customer-gateway-hystrix`.

Firstly, we are going to add Hystrix dependency to the `pom.xml`:

```
(...)

<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
    <version>${version.hystrix}</version>
</dependency>

(...)
```

The circuit breaker command is implemented by extending the `com.netflix.hystrix.HystrixCommand` class. Let's take a look at its usage at the concrete example of our `PricingProxy`:

```
(...)

@ApplicationScoped
public class PricingProxy {

    (...)

    //1
    private class GetPriceCommand extends HystrixCommand<Response> {

private final String itemId;

        //2
        public GetPriceCommand(String itemId) {
            //3
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("pricing-service"));
            this.itemId = itemId;
        }

        @Override
        //4
        protected Response run() {
            //5
            Client client = ClientBuilder.newClient();
            WebTarget target = client.target(targetPath + "/price/" + itemId);
            return target.request(MediaType.APPLICATION_JSON).get();
        }
    }
}
```

The command has to extend the abstract `HystrixCommand` class (1). The class has to be parameterized with a type that will be returned as the command result. In our example, it will be the JAX-RS response class—the same as we used in our raw invocation.

The class takes the itemId parameter as an argument (2), which will be used in the invocation.

As you can see in the constructor code, we have provided `HystrixCommandGroupKey` parameter to it (3). `HystrixCommand` constructors allow you to provide a combination of three arguments: `HystrixCommandGroupKey`, `HystrixThreadPoolKey`, and the timeout. The two enum parameters are used for command grouping: the commands with the same group key will belong to the same group and will be grouped together for the purpose of reporting, alerting, and monitoring. The thread pool key specifies the commands that belong to the same Hystrix thread pool used by the bulkhead algorithm. When the thread pool key is not enabled, the group key is used as the thread pool identifier.

As a result, in our example all the `pricingService` invocation commands will belong to the same group and will use their own thread pool. The third argument in the `HystrixCommand&nbsp;`constructors is the timeout of the invocation. If one is not provided, the default timeout is used.

We have to extend the `run` method of the HystrixCommand class (4). This method will be invoked when the command is executed. As you can see (5), the content of the method is the same as the code of the proxy in our raw invocation.

Now, let's take a look at how to execute the command:

```
(...)

@ApplicationScoped
public class PricingProxy {

    private final String targetPath = System.getProperty("proxy.pricing.url");

    //1
    public Price getPrice(String itemId){
        //2
        return new GetPriceCommand(itemId).execute().readEntity(Price.class);
    }

    (...)
}
```

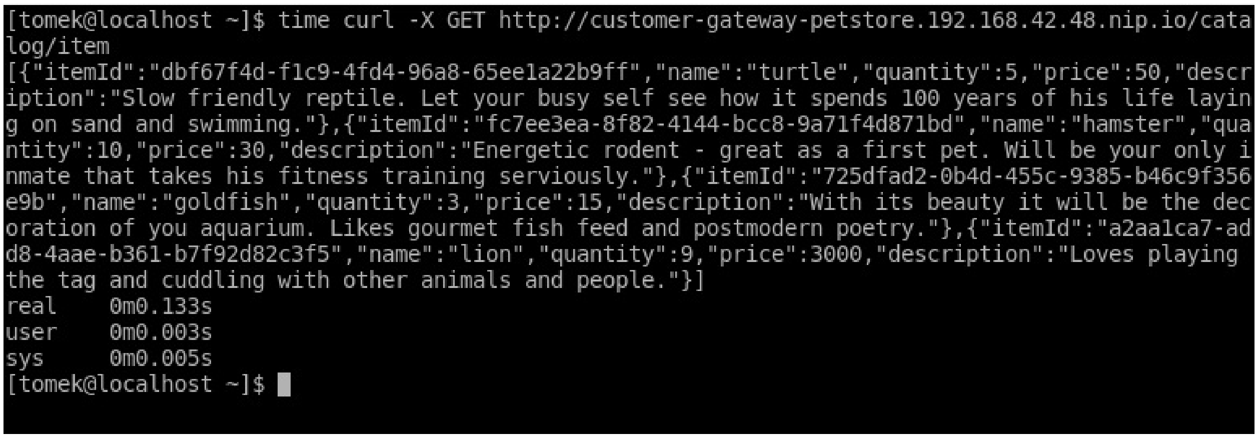
`oc delete all -l app=customer-gateway`

```
oc new-app wildflyswarm-10-centos7~https://github.com/PacktPublishing/Hands-On-Cloud-Development-with-WildFly.git \
--context-dir=chapter11/customer-gateway-hystrix --name=customer-gateway
```

`oc expose svc/customer-gateway`

`curl customer-gateway-petstore.<update-me>-80-<update-me>.environments.katacoda.com/catalog/item`

The getPrice method of the proxy (1) creates the command object (2) and invokes the `execute()` method on it. This results in the execution of the w hole circuit breaker algorithm described in the theoretical section. Let's invoke the catalog/item method now and measure the time of its invocation:



```
[tomek@localhost ~]$ time curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/cata
log/item
[{"itemId":"dbf67f4d-f1c9-4fd4-96a8-65ee1a22b9ff","name":"turtle","quantity":5,"price":50,"descr
iption":"Slow friendly reptile. Let your busy self see how it spends 100 years of his life layin
g on sand and swimming."},{"itemId":"fc7ee3ea-8f82-4144-bcc8-9a71f4d871bd","name":"hamster","qua
ntity":10,"price":30,"description":"Energetic rodent - great as a first pet. Will be your only i
nmate that takes his fitness training serviously."},{"itemId":"725dfad2-0b4d-455c-9385-b46c9f356
e9b","name":"goldfish","quantity":3,"price":15,"description":"With its beauty it will be the dec
oration of you aquarium. Likes gourmet fish feed and postmodern poetry."},{"itemId":"a2aa1ca7-ad
d8-4aae-b361-b7f92d82c3f5","name":"lion","quantity":9,"price":3000,"description":"Loves playing
the tag and cuddling with other animals and people."}]
real    0m0.133s
user    0m0.003s
sys     0m0.005s
[tomek@localhost ~]$
```

There is no difference; the invocation is executed immediately w ithout any errors. Now, let's make `pricingService` artificially unresponsive.

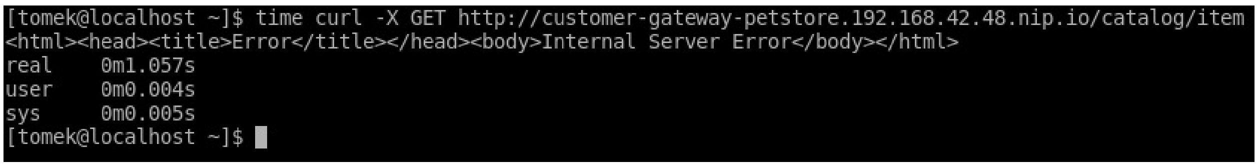
## Note

Examples reference: `chapter11/pricing-service-misbehave`.

In order to do that w e are going to make the service w ait a specified amount of time before returning a result:

```
package org.packt.swarm.petstore.pricing;

import org.packt.swarm.petstore.pricing.model.Price;

import javax.enterprise.context.ApplicationScoped;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.LockSupport;

@ApplicationScoped
public class PricingService {

@PersistenceContext(unitName = "PricingPU")
private EntityManager em;

    public Price findByItemId(String itemId) {
        //1
        LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(5));
        return em.createNamedQuery("Price.findByItemId", Price.class).setParameter("itemId", itemId).getSingleResult();
}
}
```

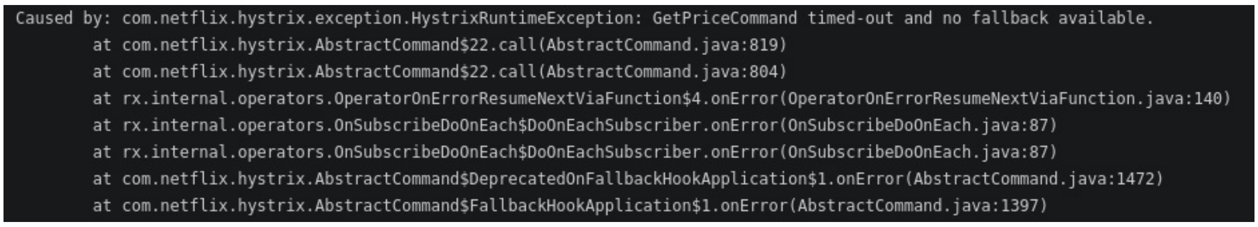
Let's deploy the new service to the cloud and retry an invocation. The result is as follows:



```
[tomek@localhost ~]$ time curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/catalog/item
<html><head><title>Error</title></head><body>Internal Server Error</body></html>
real    0m1.057s
user    0m0.004s
sys     0m0.005s
[tomek@localhost ~]$
```

As you will have noticed in the preceding screenshot, the invocation has resulted in failure. The circuit breaker was open, and there was a thread available in the thread pool. As a result, the invocation was performed, but it exceeded the default Hystrix timeout which equals 1 second.

To confirm this, let's look at the logs:



```
Caused by: com.netflix.hystrix.exception.HystrixRuntimeException: GetPriceCommand timed-out and no fallback available.
        at com.netflix.hystrix.AbstractCommand$22.call(AbstractCommand.java:819)
        at com.netflix.hystrix.AbstractCommand$22.call(AbstractCommand.java:804)
        at rx.internal.operators.OperatorOnErrorResumeNextViaFunction$4.onError(OperatorOnErrorResumeNextViaFunction.java:140)
        at rx.internal.operators.OnSubscribeDoOnEach$DoOnEachSubscriber.onError(OnSubscribeDoOnEach.java:87)
        at rx.internal.operators.OnSubscribeDoOnEach$DoOnEachSubscriber.onError(OnSubscribeDoOnEach.java:87)
        at com.netflix.hystrix.AbstractCommand$DeprecatedOnFallbackHookApplication$1.onError(AbstractCommand.java:1472)
        at com.netflix.hystrix.AbstractCommand$FallbackHookApplication$1.onError(AbstractCommand.java:1397)
```

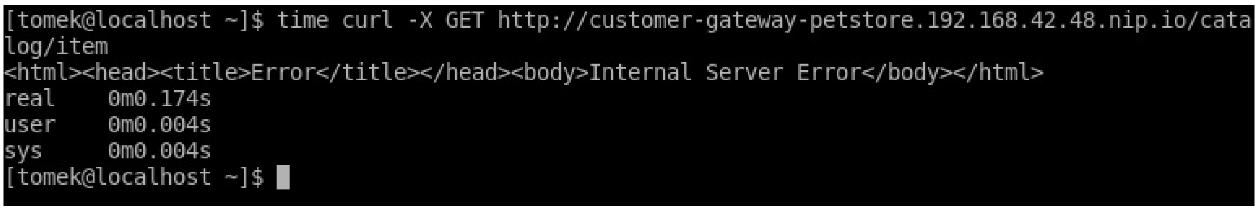
Hystrix does not lie here: the timeout was exceeded and we have implemented no fallback. We will do in a second, but before that let's learn how to modify Hystrix properties.

If you want to modify the configuration of `HystrixCommand`, you have to use the constructor with the `Setter` argument. This class allows you to configure all the constructor arguments described previously. Apart from that, the class allows you to provide the configuration properties for different aspects of the circuit breaker behavior. The exhaustive list of such properties is described in the Hystrix documentation. Here, we will present a few example modifications. Let's start with the circuit breaker timeout:

```
(...)
private class GetPriceCommand extends HystrixCommand<Response> {

private final String itemId;

    public GetPriceCommand(String itemId) {
        //1
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("pricing-service"))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
        //2
                        .withExecutionTimeoutInMilliseconds(100)));
        this.itemId = itemId;
}

@Override
protected Response run() {
        Client client = ClientBuilder.newClient();
WebTarget target = client.target(targetPath + "/price/" + itemId);
```

```
        return target.request(MediaType.APPLICATION_JSON).get();
}
}
(...)
```

The preceding code modifies our command class in order to shorten the invocation timeout to 500 milliseconds. The `Setter` class is used, and the `CommandGroupKey` is set in the same w ay as in the examples before (1). In order to modify the configuration, w e have added `HystrixCommandProperites.Setter` w ith the appropriate configuration (2). Now, let's take a look at the follow ing result:



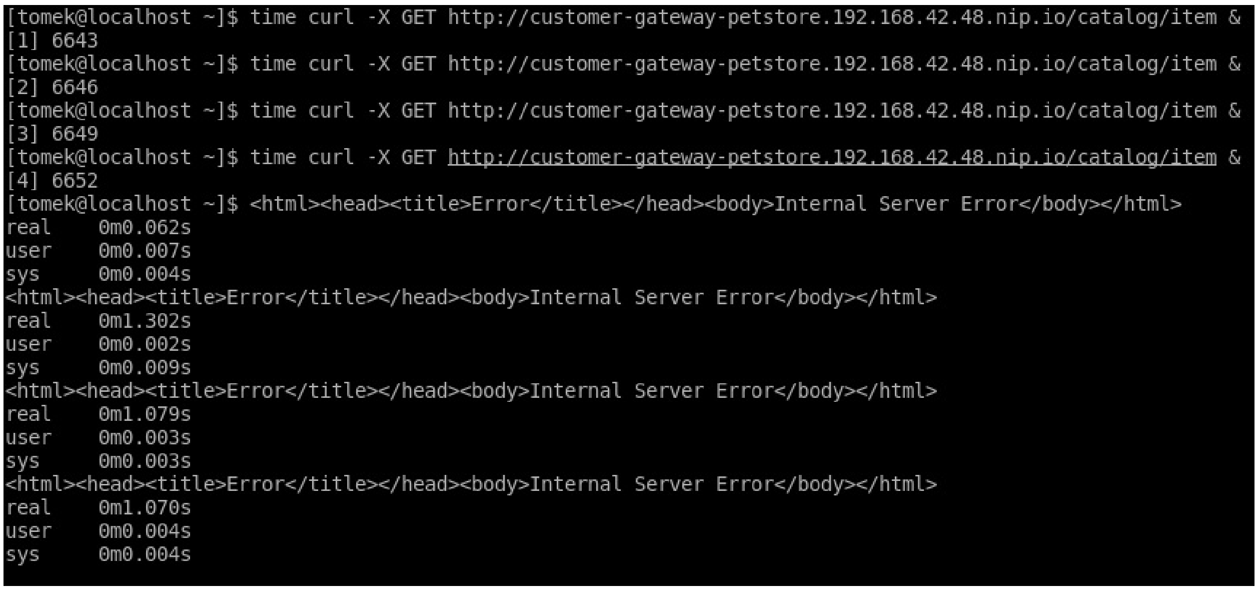
```
[tomek@localhost ~]$ time curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/cata
log/item
<html><head><title>Error</title></head><body>Internal Server Error</body></html>
real    0m0.174s
user    0m0.004s
sys     0m0.004s
[tomek@localhost ~]$
```

Let's reconfigure the application to note the behavior of the bulkheading algorithm; w e w ill increase the timeout and reduce the number of threads:

```
(...)private class GetPriceCommand extends HystrixCommand<Response> {

private final String itemId;

    public GetPriceCommand(String itemId) {
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("pricing-service"))
//1.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Setter().withCoreSize(3));
        this.itemId = itemId;
}

@Override
protected Response run() {
        Client client = ClientBuilder.newClient();
WebTarget target = client.target(targetPath + "/price/" + itemId);
        return target.request(MediaType.APPLICATION_JSON).get();
}
}
(...)
```

In order to do that, another setter (this time, the `HystrixThreadPoolProperties` setter) has to be created (1).

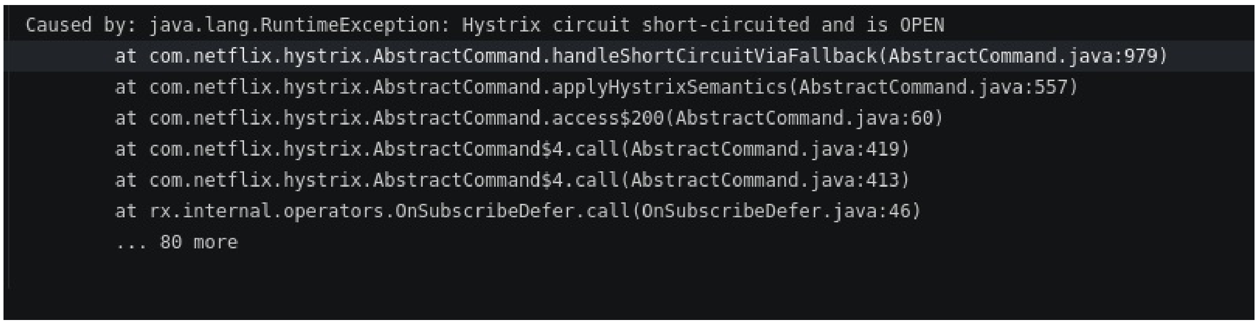
The result is as follow s:



```
[tomek@localhost ~]$ time curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/catalog/item &
[1] 6643
[tomek@localhost ~]$ time curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/catalog/item &
[2] 6646
[tomek@localhost ~]$ time curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/catalog/item &
[3] 6649
[tomek@localhost ~]$ time curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/catalog/item &
[4] 6652
[tomek@localhost ~]$ <html><head><title>Error</title></head><body>Internal Server Error</body></html>
real    0m0.062s
user    0m0.007s
sys     0m0.004s
<html><head><title>Error</title></head><body>Internal Server Error</body></html>
real    0m1.302s
user    0m0.002s
sys     0m0.009s
<html><head><title>Error</title></head><body>Internal Server Error</body></html>
real    0m1.079s
user    0m0.003s
sys     0m0.003s
<html><head><title>Error</title></head><body>Internal Server Error</body></html>
real    0m1.070s
user    0m0.004s
sys     0m0.004s
```

As you w ill have noticed in the preceding screenshot, the first three invocations have obtained their threads and w ere blocked. The fourth thread returned immediately as there are no more threads in the thread pool.

Finally, let's open the circuit. If w e run the code in a bash loop and look into the log, w e w ill note the follow ing result:



```
 Caused by: java.lang.RuntimeException: Hystrix circuit short-circuited and is OPEN
        at com.netflix.hystrix.AbstractCommand.handleShortCircuitViaFallback(AbstractCommand.java:979)
        at com.netflix.hystrix.AbstractCommand.applyHystrixSemantics(AbstractCommand.java:557)
        at com.netflix.hystrix.AbstractCommand.access$200(AbstractCommand.java:60)
        at com.netflix.hystrix.AbstractCommand$4.call(AbstractCommand.java:419)
        at com.netflix.hystrix.AbstractCommand$4.call(AbstractCommand.java:413)
        at rx.internal.operators.OnSubscribeDefer.call(OnSubscribeDefer.java:46)
        ... 80 more
```

Finally, let's implement the fallback:

```
(...)
private class CreatePaymentCommand extends HystrixCommand<Response> {

private final Payment payment;

    public CreatePaymentCommand(Payment payment) {
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey(SERVICE_NAME))
                              .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                                      .withExecutionTimeoutInMilliseconds(100)));
        this.payment = payment;
}

@Override
protected Response run() {
        Client client = ClientBuilder.newClient();
WebTarget target = client.target(targetPath + "/payment");
        return target.request(MediaType.APPLICATION_JSON).post(Entity.json(payment));
}

    @Override
    //1
    protected Response getFallback() {
//2
```

```
        return Response.status(Response.Status.SERVICE_UNAVAILABLE).build();
    }
}

(...)
```

In order to implement the fallback, you have to override the `getFallback` method (1). In our example, w e have returned the `SERVICE_UNAVAILABLE` exception w henever the `paymentService` is unreachable (2).

We can now reimplement the `PetstoreService` so that it creates a meaningful exception w henever such situations occur:

```
public String buy(int customerId){
    Cart cart = cartProxy.getCart(customerId);

Order order = createOrderFromCart(customerId, cart);
    int orderId  = orderProxy.createOrder(order);

Payment payment = new Payment();
payment.setMerchantId(Constants.MERCHANT_ID);
payment.setDescription(String.format("ORDER_ID: %s", orderId));
payment.setAmount(order.getPrice());

Response response =  paymentProxy.createPayment(payment);

    if(response.getStatus() == Response.Status.SERVICE_UNAVAILABLE.getStatusCode()){
        throw new RuntimeException("Payment service unreachable");
    }

return (String) response.readEntity(String.class);
}
```

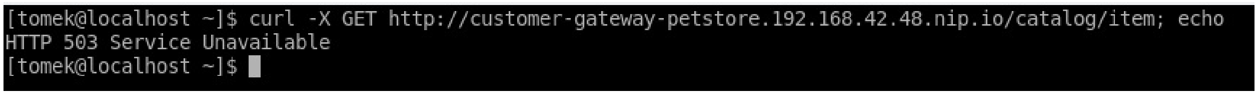
`oc delete all -l app=pricing-service`

```
oc new-app wildflyswarm-10-centos7~https://github.com/PacktPublishing/Hands-On-Cloud-Development-with-WildFly.git \
--context-dir=chapter11/pricing-service-misbehave --name=pricing-service
```

`oc expose svc/pricing-service`

`curl -X GET customer-gateway-petstore.<update-me>-80-<update-me>.environments.katacoda.com/catalog/item`

This w ill be the result of an invocation:



```
[tomek@localhost ~]$ curl -X GET http://customer-gateway-petstore.192.168.42.48.nip.io/catalog/item; echo
HTTP 503 Service Unavailable
[tomek@localhost ~]$
```