How To Use

David Gross edited this page on 27 Mar · 35 revisions

Contents

Hello World!

The following is a basic “Hello World” implementation of a `HystrixCommand`:

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

    private final String name;

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

    @Override
    protected String run() {
        // a real example would do work like a network call here
        return "Hello " + name + "!";
```

```
    }
}
```

View Source

`HystrixObservableCommand` Equivalent

An equivalent Hello World solution that uses a `HystrixObservableCommand` instead of a `HystrixCommand` would involve overriding the `construct` method as follows:

```
public class CommandHelloWorld extends HystrixObservableCommand<String> {

    private final String name;

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

    @Override
    protected Observable<String> construct() {
        return Observable.create(new Observable.OnSubscribe<String>() {
            @Override
            public void call(Subscriber<? super String> observer) {
                try {
                    if (!observer.isUnsubscribed()) {
                        // a real example would do work like a network call h
                        observer.onNext("Hello");
                        observer.onNext(name + "!");
                        observer.onCompleted();
                    }
                } catch (Exception e) {
                    observer.onError(e);
                }
            }
         } );
    }
}
```

Synchronous Execution

You can execute a `HystrixCommand` synchronously with the `execute()` method, as in the following example:

```
String s = new CommandHelloWorld("World").execute();
```

Execution of this form passes the following tests:

```
        @Test
        public void testSynchronous() {
            assertEquals("Hello World!", new CommandHelloWorld("World").execu
            assertEquals("Hello Bob!", new CommandHelloWorld("Bob").execute()
        }
```

`HystrixObservableCommand` Equivalent

There is no simple equivalent to `execute` for a `HystrixObservableCommand`, but if you

know that the `Observable` produced by such a command must always produce only a single value, you can mimic the behavior of `execute` by applying `.toBlocking().toFuture().get()` to the `Observable`.

Asynchronous Execution

You can execute a `HystrixCommand` asynchronously by using the `queue()` method, as in the following example:

```
Future<String> fs = new CommandHelloWorld("World").queue();
```

You can retrieve the result of the command by using the Future:

```
String s = fs.get();
```

The following unit tests demonstrate the behavior:

```
@Test
public void testAsynchronous1() throws Exception {
    assertEquals("Hello World!", new CommandHelloWorld("World").queue
    assertEquals("Hello Bob!", new CommandHelloWorld("Bob").queue().g
}

@Test
public void testAsynchronous2() throws Exception {

    Future<String> fWorld = new CommandHelloWorld("World").queue();
    Future<String> fBob = new CommandHelloWorld("Bob").queue();

    assertEquals("Hello World!", fWorld.get());
    assertEquals("Hello Bob!", fBob.get());
}
```

The following are equivalent to each other:

```
String s1 = new CommandHelloWorld("World").execute();
String s2 = new CommandHelloWorld("World").queue().get();
```

`HystrixObservableCommand` Equivalent

There is no simple equivalent to `queue` for a `HystrixObservableCommand`, but if you know that the `Observable` produced by such a command must always produce only a single value, you can mimic the behavior of `queue` by applying the RxJava operators `.toBlocking().toFuture()` to the `Observable`.

Reactive Execution

You can also observe the results of a `HystrixCommand` as an `Observable` by using one of the following methods:

- `observe()` — returns a “hot” Observable that executes the command immediately, though because the Observable is filtered through a `ReplaySubject` you are not in

danger of losing any items that it emits before you have a chance to subscribe

- `toObservable()` — returns a "cold" Observable that won't execute the command and begin emitting its results until you subscribe to the Observable

```
Observable<String> ho = new CommandHelloWorld("World").observe();
// or Observable<String> co = new CommandHelloWorld("World").toObservable();
```

You then retrieve the value of the command by subscribing to the Observable:

```
ho.subscribe(new Action1<String>() {

    @Override
    public void call(String s) {
         // value emitted here
    }

});
```

The following unit tests demonstrate the behavior:

```
@Test
public void testObservable() throws Exception {

    Observable<String> fWorld = new CommandHelloWorld("World").observe();
    Observable<String> fBob = new CommandHelloWorld("Bob").observe();

    // blocking
    assertEquals("Hello World!", fWorld.toBlockingObservable().single());
    assertEquals("Hello Bob!", fBob.toBlockingObservable().single());

    // non-blocking
    // - this is a verbose anonymous inner-class approach and doesn't do asse
    fWorld.subscribe(new Observer<String>() {

        @Override
        public void onCompleted() {
            // nothing needed here
        }

        @Override
        public void onError(Throwable e) {
            e.printStackTrace();
        }

        @Override
        public void onNext(String v) {
            System.out.println("onNext: " + v);
        }

    });

    // non-blocking
    // - also verbose anonymous inner-class
    // - ignore errors and onCompleted signal
    fBob.subscribe(new Action1<String>() {

        @Override
        public void call(String v) {
            System.out.println("onNext: " + v);
        }
```

```
        });
    }
```

Using Java 8 lambdas/closures is more compact; it would look like this:

```
    fWorld.subscribe((v) -> {
        System.out.println("onNext: " + v);
    })

    // - or while also including error handling

    fWorld.subscribe((v) -> {
        System.out.println("onNext: " + v);
    }, (exception) -> {
        exception.printStackTrace();
    })
```

More information about Observable can be found at http://reactivex.io/documentation/observable.html

Reactive Commands

Rather than converting a `HystrixCommand` into an `Observable` using the methods described above, you can also create a `HystrixObservableCommand` that is a specialized version of `HystrixCommand` meant to wrap Observables. A `HystrixObservableCommand` is capable of wrapping Observables that emit multiple items, whereas ordinary `HystrixCommands`, even when converted into Observables, will never emit more than one item.

In such a case, instead of overriding the `run` method with your command logic (as you would with an ordinary `HystrixCommand`), you would override the `construct` method so that it returns the Observable you intend to wrap.

To obtain an Observable representation of the `HystrixObservableCommand`, use one of the following two methods:

- `observe()` — returns a "hot" Observable that subscribes to the underlying Observable immediately, though because it is filtered through a `ReplaySubject` you are not in danger of losing any items that it emits before you have a chance to subscribe to the resulting Observable
- `toObservable()` — returns a "cold" Observable that won't subscribe to the underlying Observable until you subscribe to the resulting Observable

Fallback

You can support graceful degradation in a Hystrix command by adding a fallback method that Hystrix will call to obtain a default value or values in case the main command fails. You will want to implement a fallback for most Hystrix commands that might conceivably fail, with a couple of exceptions:

1. a command that performs a write operation
    - If your Hystrix command is designed to do a write operation rather than to return a value (such a command might normally return a `void` in the case of a

`HystrixCommand` or an empty Observable in the case of a `HystrixObservableCommand`), there isn't much point in implementing a fallback. If the write fails, you probably want the failure to propagate back to the caller.

2. batch systems/offline compute
    - If your Hystrix command is filling up a cache, or generating a report, or doing any sort of offline computation, it's usually more appropriate to propagate the error back to the caller who can then retry the command later, rather than to send the caller a silently-degraded response.

Whether or not your command has a fallback, all of the usual Hystrix state and circuit-breaker state/metrics are updated to indicate the command failure.

In an ordinary `HystrixCommand` you implement a fallback by means of a `getFallback()` implementation. Hystrix will execute this fallback for all types of failure such as `run()` failure, timeout, thread pool or semaphore rejection, and circuit-breaker short-circuiting. The following example includes such a fallback:

```
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 + "!";
    }
}
```

View Source

This command's `run()` method will fail on every execution. However, the caller will always receive the value returned by the command's `getFallback()` method instead of receiving an exception:

```
    @Test
    public void testSynchronous() {
        assertEquals("Hello Failure World!", new CommandHelloFailure("World")
        assertEquals("Hello Failure Bob!", new CommandHelloFailure("Bob").exe
    }
```

`HystrixObservableCommand` Equivalent

For a `HystrixObservableCommand` you instead may override the `resumeWithFallback` method so that it returns a second `Observable` that will take over from the primary `Observable` if it fails. Note that because an `Observable` may fail after having already emitted one or more items, your fallback should not assume that it will be emitting the only values that the observer will see.

Internally, Hystrix uses the RxJava `onErrorResumeNext` operator to seamlessly transition between the primary and fallback `Observable` in case of an error.

Error Propagation

All exceptions thrown from the `run()` method except for `HystrixBadRequestException` count as failures and trigger `getFallback()` and circuit-breaker logic.

You can wrap the exception that you would like to throw in `HystrixBadRequestException` and retrieve it via `getCause()`. The `HystrixBadRequestException` is intended for use cases such as reporting illegal arguments or non-system failures that should not count against the failure metrics and should not trigger fallback logic.

`HystrixObservableCommand` Equivalent

In the case of a `HystrixObservableCommand`, non-recoverable errors are returned via `onError` notifications from the resulting `Observable`, and fallbacks are accomplished by falling back to a second Observable that Hystrix obtains through the `resumeWithFallback` method that you implement.

Command Name

A command name is, by default, derived from the class name:

```
getClass().getSimpleName();
```

To explicitly define the name pass it in via the `HystrixCommand` or `HystrixObservableCommand` constructor:

```
    public CommandHelloWorld(String name) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Examp
                .andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))
        this.name = name;
    }
```

`HystrixCommandKey` is an interface and can be implemented as an enum or regular class, but it also has the helper `Factory` class to construct and intern instances such as:

```
HystrixCommandKey.Factory.asKey("HelloWorld")
```

Command Group

Hystrix uses the command group key to group together commands such as for reporting, alerting, dashboards, or team/library ownership.

By default Hystrix uses this to define the command thread-pool unless a separate one is defined.

`HystrixCommandGroupKey` is an interface and can be implemented as an enum or regular class, but it also has the helper `Factory` class to construct and intern instances such as:

```
HystrixCommandGroupKey.Factory.asKey("ExampleGroup")
```

Command Thread-Pool

The thread-pool key represents a `HystrixThreadPool` for monitoring, metrics publishing, caching, and other such uses. A `HystrixCommand` is associated with a single `HystrixThreadPool` as retrieved by the `HystrixThreadPoolKey` injected into it, or it defaults to one created using the `HystrixCommandGroupKey` it is created with.

To explicitly define the name pass it in via the `HystrixCommand` or `HystrixObservableCommand` constructor:

```
    public CommandHelloWorld(String name) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Examp
                .andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))
                .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("HelloWo
        this.name = name;
    }
```

`HystrixThreadPoolKey` is an interface and can be implemented as an enum or regular class, but it also has the helper `Factory` class to construct and intern instances such as:

```
HystrixThreadPoolKey.Factory.asKey("HelloWorldPool")
```

The reason why you might use `HystrixThreadPoolKey` instead of just a different `HystrixCommandGroupKey` is that multiple commands may belong to the same “group” of ownership or logical functionality, but certain commands may need to be isolated from each other.

Here is a simple example:

- two commands used to access Video metadata
- group name is “VideoMetadata”
- command A goes against resource #1
- command B goes against resource #2

If command A becomes latent and saturates its thread-pool it should not prevent command B from executing requests since they each hit different back-end resources.

Thus, we logically want these commands grouped together but want them isolated differently and would use `HystrixThreadPoolKey` to give each of them a different thread-pool.

Request Cache

You enable request caching by implementing the `getCacheKey()` method on a `HystrixCommand` or `HystrixObservableCommand` object as follows:

```
public class CommandUsingRequestCache extends HystrixCommand<Boolean> {

    private final int value;
```

```
    protected CommandUsingRequestCache(int value) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.value = value;
    }

    @Override
    protected Boolean run() {
        return value == 0 || value % 2 == 0;
    }

    @Override
    protected String getCacheKey() {
        return String.valueOf(value);
    }
}
```

View Source

Since this depends on request context we must initialize the `HystrixRequestContext`.

In a simple unit test you could do this as follows:

```
@Test
public void testWithoutCacheHits() {
    HystrixRequestContext context = HystrixRequestContext.initialize(
    try {
        assertTrue(new CommandUsingRequestCache(2).execute());
        assertFalse(new CommandUsingRequestCache(1).execute());
        assertTrue(new CommandUsingRequestCache(0).execute());
        assertTrue(new CommandUsingRequestCache(58672).execute());
    } finally {
        context.shutdown();
    }
}
```

Typically this context will be initialized and shut down via a `ServletFilter` that wraps a user request or some other lifecycle hook.

The following is an example that shows how commands retrieve their values from the cache (and how you can query an object to know whether its value came from the cache) within a request context:

```
@Test
public void testWithCacheHits() {
    HystrixRequestContext context = HystrixRequestContext.initialize(
    try {
        CommandUsingRequestCache command2a = new CommandUsingRequest(
        CommandUsingRequestCache command2b = new CommandUsingRequest(

        assertTrue(command2a.execute());
        // this is the first time we've executed this command with
        // the value of "2" so it should not be from cache
        assertFalse(command2a.isResponseFromCache());

        assertTrue(command2b.execute());
        // this is the second time we've executed this command with
        // the same value so it should return from cache
        assertTrue(command2b.isResponseFromCache());
    } finally {
        context.shutdown();
```

```
        }

        // start a new request context
        context = HystrixRequestContext.initializeContext();
        try {
            CommandUsingRequestCache command3b = new CommandUsingRequestC
            assertTrue(command3b.execute());
            // this is a new request context so this
            // should not come from cache
            assertFalse(command3b.isResponseFromCache());
        } finally {
            context.shutdown();
        }
    }
```

Request Collapsing

Request collapsing enables multiple requests to be batched into a single `HystrixCommand` instance execution.

A collapser can use the batch size and the elapsed time since the creation of the batch as triggers for executing a batch.

Following is a simple example of how to implement a `HystrixCollapser`:

```
public class CommandCollapserGetValueForKey extends HystrixCollapser<List<Str

    private final Integer key;

    public CommandCollapserGetValueForKey(Integer key) {
        this.key = key;
    }

    @Override
    public Integer getRequestArgument() {
        return key;
    }

    @Override
    protected HystrixCommand<List<String>> createCommand(final Collection<Col
        return new BatchCommand(requests);
    }

    @Override
    protected void mapResponseToRequests(List<String> batchResponse, Collecti
        int count = 0;
        for (CollapsedRequest<String, Integer> request : requests) {
            request.setResponse(batchResponse.get(count++));
        }
    }

    private static final class BatchCommand extends HystrixCommand<List<Strin
        private final Collection<CollapsedRequest<String, Integer>> requests;

        private BatchCommand(Collection<CollapsedRequest<String, Integer>> re
                super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKe
                    .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueF
            this.requests = requests;
        }

        @Override
        protected List<String> run() {
```

```
            ArrayList<String> response = new ArrayList<String>();
            for (CollapsedRequest<String, Integer> request : requests) {
                // artificial response for each argument received in the batc
                response.add("ValueForKey: " + request.getArgument());
            }
            return response;
        }
    }
}
```

View Source

The following unit test shows how to use a collapser to automatically batch four executions of `CommandCollapserGetValueForKey` into a single `HystrixCommand` execution:

```
@Test
public void testCollapser() throws Exception {
    HystrixRequestContext context = HystrixRequestContext.initializeContext()
    try {
        Future<String> f1 = new CommandCollapserGetValueForKey(1).queue();
        Future<String> f2 = new CommandCollapserGetValueForKey(2).queue();
        Future<String> f3 = new CommandCollapserGetValueForKey(3).queue();
        Future<String> f4 = new CommandCollapserGetValueForKey(4).queue();

        assertEquals("ValueForKey: 1", f1.get());
        assertEquals("ValueForKey: 2", f2.get());
        assertEquals("ValueForKey: 3", f3.get());
        assertEquals("ValueForKey: 4", f4.get());

        // assert that the batch command 'GetValueForKey' was in fact
        // executed and that it executed only once
        assertEquals(1, HystrixRequestLog.getCurrentRequest().getExecutedComm
        HystrixCommand<?> command = HystrixRequestLog.getCurrentRequest().get
        // assert the command is the one we're expecting
        assertEquals("GetValueForKey", command.getCommandKey().name());
        // confirm that it was a COLLAPSED command execution
        assertTrue(command.getExecutionEvents().contains(HystrixEventType.COL
        // and that it was successful
        assertTrue(command.getExecutionEvents().contains(HystrixEventType.SUC
    } finally {
        context.shutdown();
    }
}
```

Request Context Setup

To use request-scoped features (request caching, request collapsing, request log) you must manage the `HystrixRequestContext` lifecycle (or implement an alternative `HystrixConcurrencyStrategy`).

This means that you must execute the following before a request:

```
HystrixRequestContext context = HystrixRequestContext.initializeContext();
```

and then this at the end of the request:

```
context.shutdown();
```

In a standard Java web application, you can use a Servlet Filter to initialize this lifecycle by implementing a filter similar to this:

```
public class HystrixRequestContextServletFilter implements Filter {

    public void doFilter(ServletRequest request, ServletResponse response, Fi
     throws IOException, ServletException {
        HystrixRequestContext context = HystrixRequestContext.initializeConte
        try {
            chain.doFilter(request, response);
        } finally {
            context.shutdown();
        }
    }
}
```

You could enable the filter for all incoming traffic by adding a section to the `web.xml` as follows:

```
    <filter>
      <display-name>HystrixRequestContextServletFilter</display-name>
      <filter-name>HystrixRequestContextServletFilter</filter-name>
      <filter-class>com.netflix.hystrix.contrib.requestservlet.HystrixRequest
    </filter>
    <filter-mapping>
      <filter-name>HystrixRequestContextServletFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>
```

Common Patterns

In the following sections are common uses and patterns of use for `HystrixCommand` and `HystrixObservableCommand`.

Fail Fast

The most basic execution is one that does a single thing and has no fallback behavior. It will throw an exception if any type of failure occurs.

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

    private final boolean throwException;

    public CommandThatFailsFast(boolean throwException) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.throwException = throwException;
    }

    @Override
    protected String run() {
        if (throwException) {
            throw new RuntimeException("failure from CommandThatFailsFast");
        } else {
            return "success";
        }
    }
```

View Source

These unit tests show how it behaves:

```
@Test
public void testSuccess() {
    assertEquals("success", new CommandThatFailsFast(false).execute());
}

@Test
public void testFailure() {
    try {
        new CommandThatFailsFast(true).execute();
        fail("we should have thrown an exception");
    } catch (HystrixRuntimeException e) {
        assertEquals("failure from CommandThatFailsFast", e.getCause().getMes
        e.printStackTrace();
    }
}
```

`HystrixObservableCommand` Equivalent

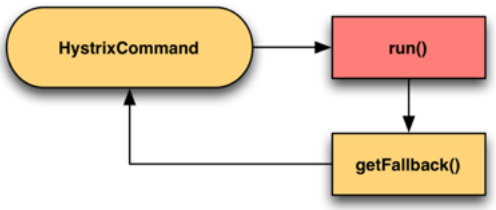
The equivalent Fail-Fast solution for a `HystrixObservableCommand` would involve overriding the `resumeWithFallback` method as follows:

```
    @Override
    protected Observable<String> resumeWithFallback() {
        if (throwException) {
            return Observable.error(new Throwable("failure from CommandThatFa
        } else {
            return Observable.just("success");
        }
    }
```

Fail Silent

Failing silently is the equivalent of returning an empty response or removing functionality. It can be done by returning `null` , an empty Map, empty List, or other such responses.

You do this by implementing a `getFallback()` method on the `HystrixCommand` instance:




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

    private final boolean throwException;

    public CommandThatFailsSilently(boolean throwException) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.throwException = throwException;
    }
```

```
    @Override
    protected String run() {
        if (throwException) {
            throw new RuntimeException("failure from CommandThatFailsFast");
        } else {
            return "success";
        }
    }

    @Override
    protected String getFallback() {
        return null;
    }
}
```

View Source

```
@Test
public void testSuccess() {
    assertEquals("success", new CommandThatFailsSilently(false).execute());
}

@Test
public void testFailure() {
    try {
        assertEquals(null, new CommandThatFailsSilently(true).execute());
    } catch (HystrixRuntimeException e) {
        fail("we should not get an exception as we fail silently with a fallb
    }
}
```

Another implementation that returns an empty list would look like:

```
    @Override
    protected List<String> getFallback() {
        return Collections.emptyList();
    }
```

`HystrixObservableCommand` Equivalent

The equivalent Fail-Silently solution for a `HystrixObservableCommand` would involve overriding the `resumeWithFallback()` method as follows:

```
    @Override
    protected Observable<String> resumeWithFallback() {
        return Observable.empty();
    }
```

Fallback: Static

Fallbacks can return default values statically embedded in code. This doesn't cause the feature or service to be removed in the way that "fail silent" often does, but instead causes default behavior to occur.

For example, if a command returns a true/false based on user credentials but the command execution fails, it can default to true:

```
    @Override
    protected Boolean getFallback() {
        return true;
    }
```

`HystrixObservableCommand` Equivalent

The equivalent Static solution for a `HystrixObservableCommand` would involve overriding the `resumeWithFallback` method as follows:

```
    @Override
    protected Observable<Boolean> resumeWithFallback() {
        return Observable.just( true );
    }
```

Fallback: Stubbed

You typically use a stubbed fallback when your command returns a compound object containing multiple fields, some of which can be determined from other request state while other fields are set to default values.

Examples of places where you might find state appropriate to use in these stubbed values are:

- cookies
- request arguments and headers
- responses from previous service requests prior to the current one failing

Your fallback can retrieve stubbed values statically from the request scope, but typically it is recommended that they be injected at command instantiation time for use if they are needed such as this following example demonstrates in the way it treats the `countryCodeFromGeoLookup` field:

```
public class CommandWithStubbedFallback extends HystrixCommand<UserAccount> {

    private final int customerId;
    private final String countryCodeFromGeoLookup;

    /**
     * @param customerId
     *            The customerID to retrieve UserAccount for
     * @param countryCodeFromGeoLookup
     *            The default country code from the HTTP request geo code loc
     */
    protected CommandWithStubbedFallback(int customerId, String countryCodeFr
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.customerId = customerId;
        this.countryCodeFromGeoLookup = countryCodeFromGeoLookup;
    }

    @Override
    protected UserAccount run() {
        // fetch UserAccount from remote service
        //        return UserAccountClient.getAccount(customerId);
        throw new RuntimeException("forcing failure for example");
    }
```

```
    @Override
    protected UserAccount getFallback() {
        /**
         * Return stubbed fallback with some static defaults, placeholders,
         * and an injected value 'countryCodeFromGeoLookup' that we'll use
         * instead of what we would have retrieved from the remote service.
         */
        return new UserAccount(customerId, "Unknown Name",
                countryCodeFromGeoLookup, true, true, false);
    }

    public static class UserAccount {
        private final int customerId;
        private final String name;
        private final String countryCode;
        private final boolean isFeatureXPermitted;
        private final boolean isFeatureYPermitted;
        private final boolean isFeatureZPermitted;

        UserAccount(int customerId, String name, String countryCode,
                boolean isFeatureXPermitted,
                boolean isFeatureYPermitted,
                boolean isFeatureZPermitted) {
            this.customerId = customerId;
            this.name = name;
            this.countryCode = countryCode;
            this.isFeatureXPermitted = isFeatureXPermitted;
            this.isFeatureYPermitted = isFeatureYPermitted;
            this.isFeatureZPermitted = isFeatureZPermitted;
        }
    }
}
```

View Source

The following unit test demonstrates its behavior:

```
    @Test
    public void test() {
        CommandWithStubbedFallback command = new CommandWithStubbedFallback(1
        UserAccount account = command.execute();
        assertTrue(command.isFailedExecution());
        assertTrue(command.isResponseFromFallback());
        assertEquals(1234, account.customerId);
        assertEquals("ca", account.countryCode);
        assertEquals(true, account.isFeatureXPermitted);
        assertEquals(true, account.isFeatureYPermitted);
        assertEquals(false, account.isFeatureZPermitted);
    }
```

`HystrixObservableCommand` Equivalent

The equivalent Stubbed solution for a `HystrixObservableCommand` would involve overriding the `resumeWithFallback` method to return an `Observable` that emits the stub responses. A version equivalent to the previous example would look like this:

```
@Override
protected Observable<Boolean> resumeWithFallback() {
    return Observable.just( new UserAccount(customerId, "Unknown Name",
                                            countryCodeFromGeoLookup, true, t
}
```

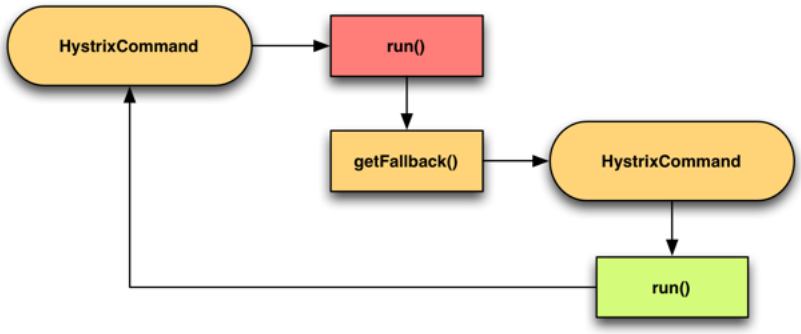
But if you are expecting to emit multiple items from your `Observable`, you may be more interested in generating stubs for only those items that the original `Observable` had not yet emitted before it failed. Here is a simple example to show how you might accomplish this — it keeps track of the last item emitted from the main `Observable` so that the fallback knows where to pick up to continue the sequence:

```
@Override
protected Observable<Integer> construct() {
    return Observable.just(1, 2, 3)
            .concatWith(Observable.<Integer> error(new RuntimeException("forc
            .doOnNext(new Action1<Integer>() {
                @Override
                public void call(Integer t1) {
                    lastSeen = t1;
                }

            })
            .subscribeOn(Schedulers.computation());
}

@Override
protected Observable<Integer> resumeWithFallback() {
    if (lastSeen < 4) {
        return Observable.range(lastSeen + 1, 4 - lastSeen);
    } else {
        return Observable.empty();
    }
}
```

Fallback: Cache via Network

Sometimes if a back-end service fails, a stale version of data can be retrieved from a cache service such as memcached.

Since the fallback will go over the network it is another possible point of failure and so it also needs to be wrapped by a `HystrixCommand` or `HystrixObservableCommand`.



It is important to execute the fallback command on a separate thread-pool, otherwise if the main command were to become latent and fill the thread-pool, this would prevent the fallback from running if the two commands share the same pool.

The following code shows how `CommandWithFallbackViaNetwork` executes `FallbackViaNetwork` in its `getFallback()` method.

Note how if the fallback fails, it *also* has a fallback which does the “fail silent” approach of

returning `null`.

To configure the `FallbackViaNetwork` command to run on a different threadpool than the default `RemoteServiceX` derived from the `HystrixCommandGroupKey`, it injects `HystrixThreadPoolKey.Factory.asKey("RemoteServiceXFallback")` into the constructor.

This means `CommandWithFallbackViaNetwork` will run on a thread-pool named `RemoteServiceX` and `FallbackViaNetwork` will run on a thread-pool named `RemoteServiceXFallback`.

```
public class CommandWithFallbackViaNetwork extends HystrixCommand<String> {
    private final int id;

    protected CommandWithFallbackViaNetwork(int id) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Remot
                .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueComma
        this.id = id;
    }

    @Override
    protected String run() {
        //        RemoteServiceXClient.getValue(id);
        throw new RuntimeException("force failure for example");
    }

    @Override
    protected String getFallback() {
        return new FallbackViaNetwork(id).execute();
    }

    private static class FallbackViaNetwork extends HystrixCommand<String> {
        private final int id;

        public FallbackViaNetwork(int id) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("F
                    .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueF
                    // use a different threadpool for the fallback command
                    // so saturating the RemoteServiceX pool won't prevent
                    // fallbacks from executing
                    .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("Rem
            this.id = id;
        }

        @Override
        protected String run() {
            MemCacheClient.getValue(id);
        }

        @Override
        protected String getFallback() {
            // the fallback also failed
            // so this fallback-of-a-fallback will
            // fail silently and return null
            return null;
        }
    }
}
```

View Source

Primary + Secondary with Fallback

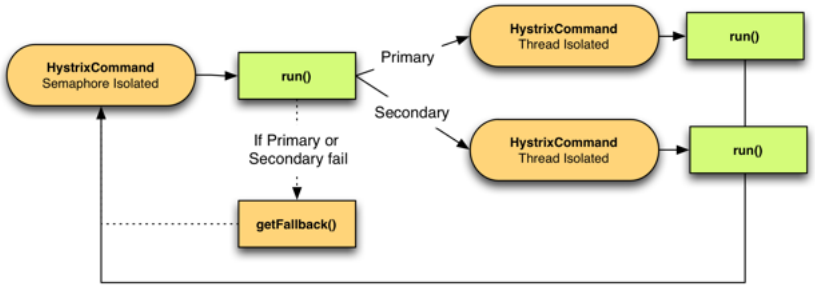
Some systems have dual-mode behavior — primary and secondary, or primary and failover.

Sometimes the secondary or failover is considered a failure state and it is intended only for fallback; in those scenarios it would fit in the same pattern as “Cache via Network” described above.

However, if flipping to the secondary system is common, such as a normal part of rolling out new code (sometimes this is part of how stateful systems handle code pushes) then every time the secondary system is used the primary will be in a failure state, tripping circuit breakers and firing alerts.

This is not the desired behavior, if for no other reason than to avoid the “cry wolf” fatigue that will cause alerts to be ignored when a real issue is occurring.

So in such a case the strategy is instead to treat the switching between primary and secondary as normal, healthy patterns and put a façade in front of them.



The primary and secondary `HystrixCommand` implementations are thread-isolated since they are doing network traffic and business logic. They may each have very different performance characteristics (often the secondary system is a static cache) so another benefit of separate commands for each is that they can be individually tuned.

You do not expose these two commands publicly but you instead hide them behind another `HystrixCommand` that is semaphore-isolated and that implements the conditional logic as to whether to invoke the primary or secondary command. If both primary and secondary fail then control switches to the fallback of the façade command itself.

The façade `HystrixCommand` can use semaphore-isolation since all of the work it is doing is going through two other `HystrixCommand` s that are already thread-isolated. It is unnecessary to have yet another layer of threading as long as the `run()` method of the façade is not doing any other network calls, retry logic, or other “error prone” things.

```
public class CommandFacadeWithPrimarySecondary extends HystrixCommand<String>

    private final static DynamicBooleanProperty usePrimary = DynamicPropertyF

    private final int id;

    public CommandFacadeWithPrimarySecondary(int id) {
        super(Setter
                .withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX")
                .andCommandKey(HystrixCommandKey.Factory.asKey("PrimarySecond
                .andCommandPropertiesDefaults(
                        // we want to default to semaphore-isolation since th
                        // 2 others commands that are already thread isolated
```

```
                    HystrixCommandProperties.Setter()
                            .withExecutionIsolationStrategy(ExecutionIso
        this.id = id;
    }

    @Override
    protected String run() {
        if (usePrimary.get()) {
            return new PrimaryCommand(id).execute();
        } else {
            return new SecondaryCommand(id).execute();
        }
    }

    @Override
    protected String getFallback() {
        return "static-fallback-" + id;
    }

    @Override
    protected String getCacheKey() {
        return String.valueOf(id);
    }

    private static class PrimaryCommand extends HystrixCommand<String> {

        private final int id;

        private PrimaryCommand(int id) {
            super(Setter
                    .withGroupKey(HystrixCommandGroupKey.Factory.asKey("Syste
                    .andCommandKey(HystrixCommandKey.Factory.asKey("PrimaryCo
                    .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("Pri
                    .andCommandPropertiesDefaults(
                            // we default to a 600ms timeout for primary
                            HystrixCommandProperties.Setter().withExecutionTi
            this.id = id;
        }

        @Override
        protected String run() {
            // perform expensive 'primary' service call
            return "responseFromPrimary-" + id;
        }

    }

    private static class SecondaryCommand extends HystrixCommand<String> {

        private final int id;

        private SecondaryCommand(int id) {
            super(Setter
                    .withGroupKey(HystrixCommandGroupKey.Factory.asKey("Syste
                    .andCommandKey(HystrixCommandKey.Factory.asKey("Secondary
                    .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("Sec
                    .andCommandPropertiesDefaults(
                            // we default to a 100ms timeout for secondary
                            HystrixCommandProperties.Setter().withExecutionTi
            this.id = id;
        }

        @Override
        protected String run() {
            // perform fast 'secondary' service call
            return "responseFromSecondary-" + id;
```

```
        }

    }

    public static class UnitTest {

        @Test
        public void testPrimary() {
            HystrixRequestContext context = HystrixRequestContext.initializeC
            try {
                ConfigurationManager.getConfigInstance().setProperty("primary
                assertEquals("responseFromPrimary-20", new CommandFacadeWithF
            } finally {
                context.shutdown();
                ConfigurationManager.getConfigInstance().clear();
            }
        }

        @Test
        public void testSecondary() {
            HystrixRequestContext context = HystrixRequestContext.initializeC
            try {
                ConfigurationManager.getConfigInstance().setProperty("primary
                assertEquals("responseFromSecondary-20", new CommandFacadeWit
            } finally {
                context.shutdown();
                ConfigurationManager.getConfigInstance().clear();
            }
        }
    }
}
```

View Source

Client Doesn't Perform Network Access

When you wrap behavior that does not perform network access, but where latency is a concern or the threading overhead is unacceptable, you can set the `executionIsolationStrategy` property to `ExecutionIsolationStrategy.SEMAPHORE` and Hystrix will use semaphore isolation instead.

The following shows how to set this property as the default for a command via code (you can also override it via dynamic properties at runtime).

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

    private final int id;

    public CommandUsingSemaphoreIsolation(int id) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Examp
                // since we're doing an in-memory cache lookup we choose SEMA
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter
                        .withExecutionIsolationStrategy(ExecutionIsolationStr
        this.id = id;
    }

    @Override
    protected String run() {
        // a real implementation would retrieve data from in memory data stru
        return "ValueFromHashMap_" + id;
    }
```

```
}
```

View Source

Get-Set-Get with Request Cache Invalidation

If you are implementing a Get-Set-Get use case where the Get receives enough traffic that request caching is desired but sometimes a Set occurs on another command that should invalidate the cache within the same request, you can invalidate the cache by calling `HystrixRequestCache.clear()`.

Here is an example implementation:

```
public class CommandUsingRequestCacheInvalidation {

    /* represents a remote data store */
    private static volatile String prefixStoredOnRemoteDataStore = "ValueBefo

    public static class GetterCommand extends HystrixCommand<String> {

        private static final HystrixCommandKey GETTER_KEY = HystrixCommandKey
        private final int id;

        public GetterCommand(int id) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("G
                    .andCommandKey(GETTER_KEY));
            this.id = id;
        }

        @Override
        protected String run() {
            return prefixStoredOnRemoteDataStore + id;
        }

        @Override
        protected String getCacheKey() {
            return String.valueOf(id);
        }

        /**
         * Allow the cache to be flushed for this object.
         *
         * @param id
         *            argument that would normally be passed to the command
         */
        public static void flushCache(int id) {
            HystrixRequestCache.getInstance(GETTER_KEY,
                    HystrixConcurrencyStrategyDefault.getInstance()).clear(St
        }

    }

    public static class SetterCommand extends HystrixCommand<Void> {

        private final int id;
        private final String prefix;

        public SetterCommand(int id, String prefix) {
            super(HystrixCommandGroupKey.Factory.asKey("GetSetGet"));
            this.id = id;
            this.prefix = prefix;
        }
```

```
        @Override
        protected Void run() {
            // persist the value against the datastore
            prefixStoredOnRemoteDataStore = prefix;
            // flush the cache
            GetterCommand.flushCache(id);
            // no return value
            return null;
        }
    }
}
```

View Source

The unit test that confirms the behavior is:

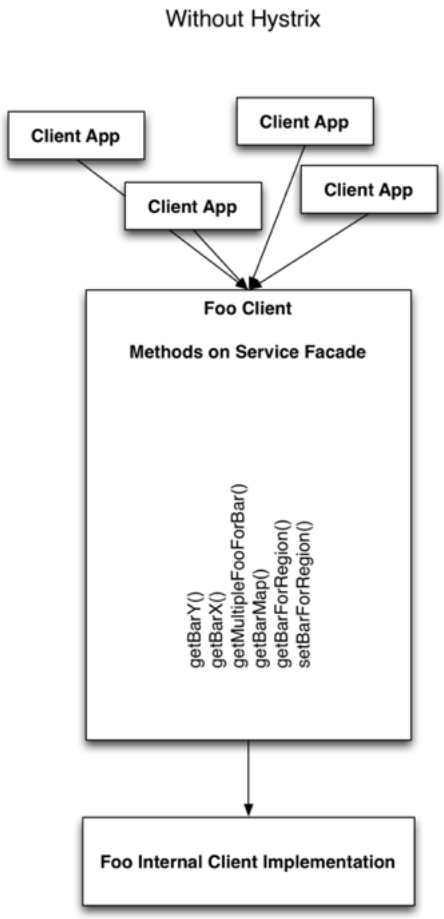
```
        @Test
        public void getGetSetGet() {
            HystrixRequestContext context = HystrixRequestContext.initializeC
            try {
                assertEquals("ValueBeforeSet_1", new GetterCommand(1).execute
                GetterCommand commandAgainstCache = new GetterCommand(1);
                assertEquals("ValueBeforeSet_1", commandAgainstCache.execute(
                // confirm it executed against cache the second time
                assertTrue(commandAgainstCache.isResponseFromCache());
                // set the new value
                new SetterCommand(1, "ValueAfterSet_").execute();
                // fetch it again
                GetterCommand commandAfterSet = new GetterCommand(1);
                // the getter should return with the new prefix, not the valu
                assertFalse(commandAfterSet.isResponseFromCache());
                assertEquals("ValueAfterSet_1", commandAfterSet.execute());
            } finally {
                context.shutdown();
            }
        }
    }
```

Migrating a Library to Hystrix

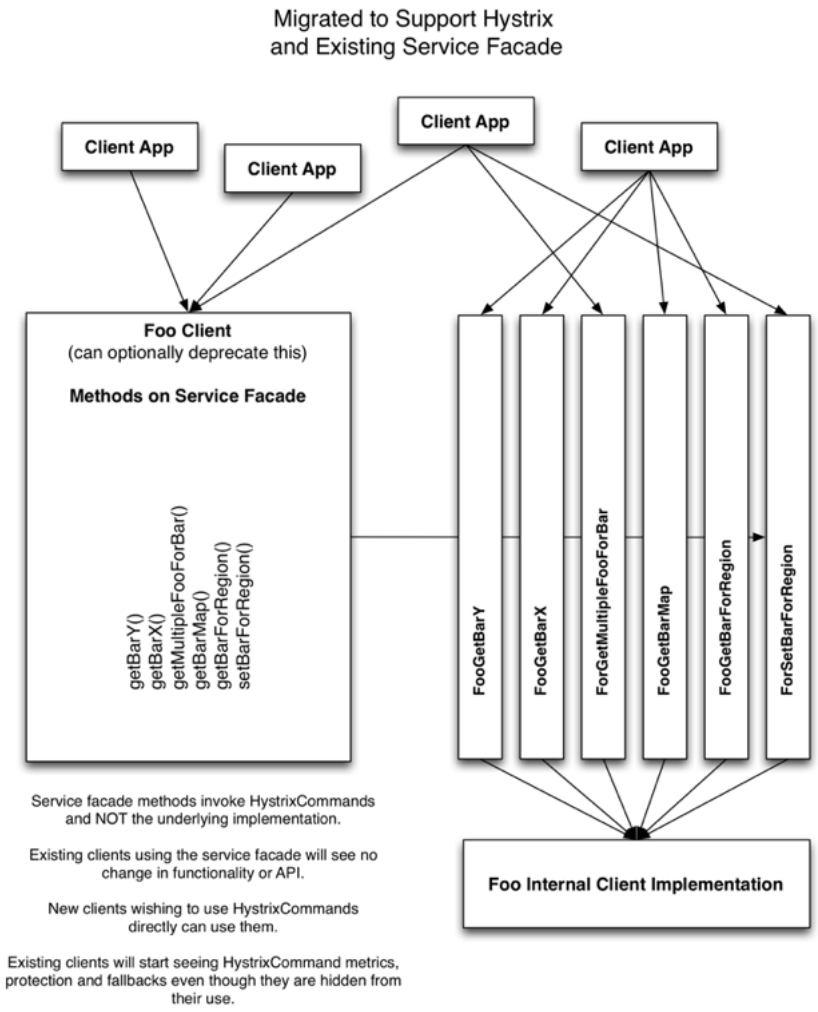
When you are migrating an existing client library to use Hystrix, you should replace each of the “service methods&rdquo with a `HystrixCommand`.

The service methods should then forward calls to the `HystrixCommand` and not have any additional business logic in them.

Thus, before migration a service library may look like this:



After migrating, users of a library will be able to access the `HystrixCommand` s directly or indirectly via the service facade that delegates to the `HystrixCommand` s.



A Netflix Original Production

Tech Blog | Twitter @NetflixOSS | Twitter @HystrixOSS | Jobs