Unit testing HTTP retry strategies

The sample code below is available in my github repo (https://github.com/madhur/unit-test-http-retry)

Let’s face it, HTTP network calls can face intermittently. Even if you best configure the Linux Operating System, the TCP/IP networking stack and the best possible private / public cloud networks, failures are inevitable. Sometimes, the reason can be dropped packets at router / NIC or could be any issue in networking stack.

For any well designed backend application, it is imperative to design for failures upfront rather than as afterthought. It depends on scenario to scenario, for example, mobile client making a call to HTTP server or it could be even a server making a inter-service call in a microservice architecture. The solution differs from scenario to scenario.

In this post, we will not discuss about possible solutions, but look at how to accurately unit test the choosen solution.

For example, a very naive HTTP retry strategy to be to retry 3 times at an interval of 500ms in case of any SocketException (https://docs.oracle.com/javase/7/docs/api/java/net/SocketException.html) or if the HTTP response status code is Service Unavailable (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503)

If we are using Java and Spring Framework (https://spring.io/), we could implement this in Apache HTTP Client (https://hc.apache.org/httpcomponents-client-ga/) as follows

```
@Bean
public HttpClient retryHttpClient() {

    connectionManager.setDefaultMaxPerRoute(CONN_POOL_DEFAULT_MAX_PER_ROUTE);
    connectionManager.setMaxTotal(CONN_POOL_DEFAULT_MAX);

    RequestConfig requestConfig = RequestConfig.custom()
            .setConnectTimeout(CONN_TIMEOUT_MS)
            .setConnectionRequestTimeout(CONN_REQUEST_TIMEOUT_MS)
            .setSocketTimeout(CONN_SOCKET_TIMEOUT_MS).build();

    return HttpClientBuilder.create()
            .setConnectionManager(connectionManager)
            .setDefaultRequestConfig(requestConfig)
            .setRetryHandler((exception, executionCount, context) -> executionCount <= MAX_RETRIES
                    && exception instanceof SocketException).setServiceUnavailableRetryStrategy(new ServiceUnavaila
bleRetryStrategy() {
                @Override
                public long getRetryInterval() {
                    return 500;
                }
                @Override
                public boolean retryRequest(HttpResponse response,
                                            int executionCount, HttpContext context) {
                    return executionCount <= MAX_RETRIES && response
                            .getStatusLine()
                            .getStatusCode() == HttpStatus.SC_SERVICE_UNAVAILABLE;
                }
            })
            .build();
}
```

But how do we go about testing that it works according to our defined specification? In this case, retrying only for SocketException or 503 status code. The strategy here is very simple but it could be complex or there could be multiple instances of HTTP clients implementing various strategies. How do we make sure they all work ? This is the question I recently faced while working with a big enterprise project recently.

Turns out, there is an excellent library called Wiremock (http://wiremock.org/) for spinning up a mock HTTP server during unit testing and it can service defined responses or response status codes. It can even generate network faults such as connection reset or network socket exceptions, which allows us to simulate any network response from the unit test code.

In addition to that, one powerful aspect of Wiremock is its Verify APIs (http://wiremock.org/docs/verifying/), which allows us to exactly measure how many times a particular endpoint was called. This is what we will be using in our tests.

To spin up a mock server, we simply add a Junit Rule (https://junit.org/junit4/javadoc/4.12/org/junit/Rule.html), within our test class

```
@Rule
public WireMockRule wireMockRule = new WireMockRule(port); // No-args constructor defaults to port 8080
```

Now, lets say we want to ensure our client retries for 503 status but not for 502 status code, we can simply test using the following tests

```
    @Test
    public void testFourRetryFor503StatusCode() {
        stubFor(get(urlEqualTo(testResource)).willReturn(aResponse().withStatus(503)));

        try {
            retryHttpClient.execute(new HttpGet(getMockUri()));
        } catch (IOException | URISyntaxException e) {
            e.printStackTrace();
            Assert.fail();
        }
        verify(1 + 3, getRequestedFor(urlEqualTo(testResource)));
    }

    @Test
    public void testZeroRetryFor502StatusCode() {
        stubFor(get(urlEqualTo(testResource)).willReturn(aResponse().withStatus(502)));

        try {
            retryHttpClient.execute(new HttpGet(getMockUri()));
        } catch (IOException | URISyntaxException e) {
            e.printStackTrace();
            Assert.fail();
        }
        verify(1, getRequestedFor(urlEqualTo(testResource)));
    }
```

Similarly, to generate a socket exception we can configure the response to generate a Fault (http://wiremock.org/docs/simulating-faults/) as follows

```
@Test
public void testFourRetryForSocketException() {
    stubFor(get(urlEqualTo(testResource)).willReturn(aResponse().withFault(Fault.CONNECTION_RESET_BY_PEER)));

    try {
        retryHttpClient.execute(new HttpGet(getMockUri()));
    } catch (IOException e) {
        // IOException is expected
    }
    catch (URISyntaxException e) {
        e.printStackTrace();
        Assert.fail();
    }
    verify(1 + 3, getRequestedFor(urlEqualTo(testResource)));
}
```

The sample code above is available in my github repo (https://github.com/madhur/unit-test-http-retry)