**How Java Developers Can Use the Wiremock Framework to Simulate HTTP-Based APIs**

|  |
| --- |
| **Key takeaways**   * Wiremock network-mocking framework is a robust time saver for creating over the wire test doubles. * It is easy to code happy path and alternate path examples using Wiremock in JUnit tests. * We will cover an example of standalone Wiremock use * We also cover how less technical testers can use the UI. * Alternatives are also discussed. |

**Introduction: My experience with over-the-wire stubs and mocks**

During my career I have had the pleasure of working at a range of large corporations, where I observed a recurring development pattern: teams are writing the same tools over and over again. I have seen not one but three over-the-wire HTTP(s) stubs/mocks ([also sometimes called service virtualization tools](https://www.infoq.com/articles/stubbing-mocking-service-virtualization-differences)) developed in-house (different code-bases) at two different banks. I have seen five of them across different teams on the same floor in a media company and at least one in an airline industry enterprise.

There was also a pattern for those HTTP stubs’ appearance and responsibilities which were applied in two main use cases:

1. In automated tests there was an interface that could be used for stubbing test doubles from JUnit tests remotely, that allowed developers to run builds in a CI (Continuous Integration) environment like TeamCity or Jenkins.
2. In a few cases, those tools also had a web interface that allowed less technical testers to do manual exploratory testing.

Since the same tool was used for both automated and manual exploratory testing, there was a lot of stub-reuse happening. Testers would use the same stubs developers created, which helped facilitate communication between them. It also helped reduce total amount of work spent on developing test doubles, because there was less duplication compared to a situation where developers and testers would use different solutions . Some teams needed only the programmatic API and no web interface because there were no testers on the team.

**Industry uses off-the-shelf tools**

Let’s view this syndrome from the wider industry perspective, with a look at logging and data storage.

If you really wanted to, you could easily implement your own simple logging framework in no more than a day or two. But conventional wisdom has conditioned us never to write logging frameworks from scratch. Rather, we use Logback, Slf4J or similar off-the-shelf logging frameworks.

And that’s for logging, a relatively simple function; if you ramp up the complexity of the problem, to consider for example data storage, it is almost a given that you will be using one of the ready-made frameworks available on the market, such as Oracle, MongoDB or Neo4J, depending on your requirements.

Or take for example testing frameworks. You would not write a testing framework for every project you start; you’d just use JUnit or any of the other ones available, and optionally build on on top of that.

In summary, there are usually off-the-shelf solutions addressing most common problem spaces.

**Wiremock – an off-the-shelf over-the-wire stub/mock/virtual service**

One candidate for a prepackaged solution is an over-the-wire [stubbing/mocking/service virtualization](https://www.infoq.com/articles/stubbing-mocking-service-virtualization-differences) solution. One such solution that has been gaining traction is Wiremock, an open source platform written in Java, that can be used in JUnit tests or as a standalone process to stub out external HTTP dependencies.

**Using Wiremock in JUnit**

Let us assume we have a weather application that talks to the forecast.io HTTP APIs to present wind speeds in London. Our application gets the weather forecast for London as a JSON string, then parses the response, extracts the current wind speed, and presents it to the user via a restful interface.

We're going to start with a JUnit rule in a test class. It's going to be a weather service mock, pretending to be the forecast.io web service:

public class WeatherApplicationTest {

@Rule

public WireMockRule forecastIoService = new WireMockRule();

}

We're going to add the weather application object, our system under test. The start and stop methods are called in setup and teardown methods. We start with a simple happy path test servesWindSpeedBasedOnForecastIoResponse:

public class WeatherApplicationTest {

@Rule

public WireMockRule forecastIoService = new WireMockRule();

private WeatherApplication weatherApplication = new WeatherApplication();

@Before

public void setUp() {

weatherApplication.start();

}

@After

public void tearDown() {

weatherApplication.stop();

}

@Test

public void servesWindSpeedBasedOnForecastIoResponse(){

// TODO

**Happy path test**

We're going to check that when we send a request to our weather application, it goes to the forecast.io APIs, and when it gets back a response, the application parses it successfully, appends a suffix to the response, and presents it to the user.

First, we start with the assertion. We're expecting to see the wind speed followed by a suffix of miles per hour:

@Test

public void servesWindSpeedBasedOnForecastIoResponse() {

assertEquals("12.34mph", content.toString());

}

But to get that, we need to talk to our application, so let us use Apache HTTP client to do that:

@Test

public void servesWindSpeedBasedOnForecastIoResponse() throws IOException {

Content content = Request.Get("http://localhost:" + weatherApplication.port() + "/wind-speed")

.execute()

.returnContent();

assertEquals("12.34mph", content.toString());

}

Now, the application will go to forecast.io APIs and try to fetch the forecast. So, we need the stub to pretend to be the real API:

@Test

public void servesWindSpeedBasedOnForecastIoResponse() throws IOException {

forecastIoService.stubFor(get(urlEqualTo("/forecast/e67b0e3784104669340c3cb089412b67/51.507253,-0.127755"))

.willReturn(aResponse().withBody("{\"currently\":{\"windSpeed\":12.34}}")));

Content content = Request.Get("http://localhost:" + weatherApplication.port() + "/wind-speed")

.execute()

.returnContent();

assertEquals("12.34mph", content.toString());

}

We tell Wiremock, represented by the forecastIoService object, to create a stub for a get request to URL  
"/forecast/e67b0e3784104669340c3cb089412b67/51.507253,-0.127755" and return a response with the given json body.

The response JSON payload returned by a real forecast.io API is about 2kb large, so to simplify this test I have included only the bits that matter. This article is about showing how to use Wiremock; it does not focus on how to do testing ([TDD](http://martinfowler.com/bliki/TestDrivenDevelopment.html) and [the testing pyramid](http://martinfowler.com/bliki/TestPyramid.html)) properly.

**Sad path test**

So now you know how a sample happy path would look like. Let us have a look at a sad path as well. Let's say we don't want to blow up with an exception whenever there's a problem with the forecast.io service. Whenever there's a problem on the forecast.io side, we're going to be presenting a 503 error to the user without blowing up, failing gracefully instead.

Again, we will be talking to the same /windspeed endpoint and stubbing forecast.io. We tell it to create the stub for a get request, and that should return a response with status code internal server error (HTTP code 500). We assert that the user will get an error message and a service unavailable (HTTP code 503) code instead of presenting a generic stacktrace:

@Test

public void reportsErrorWhenForecastIoReturnsANonSuccessfulResponse() throws IOException {

forecastIoService.stubFor(get(urlEqualTo("/forecast/e67b0e3784104669340c3cb089412b67/51.507253,-0.127755"))

.willReturn(aResponse().withStatus(SC\_INTERNAL\_SERVER\_ERROR)));

HttpResponse httpResponse = Request.Get("http://localhost:" + weatherApplication.port() + "/wind-speed")

.execute()

.returnResponse();

assertEquals(503, httpResponse.getStatusLine().getStatusCode());

assertEquals("ERROR", IOUtils.toString(httpResponse.getEntity().getContent()));

}

This is a very simple example, but it demonstrates how you can do both happy and sad paths automated testing with Wiremock.

Finally, this is how the whole test class would look like:

import com.github.tomakehurst.wiremock.junit.WireMockRule;

import org.apache.commons.io.IOUtils;

import org.apache.http.HttpResponse;

import org.apache.http.client.fluent.Content;

import org.apache.http.client.fluent.Request;

import org.junit.After;

import org.junit.Before;

import org.junit.Rule;

import org.junit.Test;

import static com.github.tomakehurst.wiremock.client.WireMock.\*;

import static javax.servlet.http.HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR;

import static junit.framework.TestCase.assertEquals;

public class WeatherApplicationTest {

@Rule

public WireMockRule forecastIoService = new WireMockRule();

private WeatherApplication weatherApplication = new WeatherApplication();

@Before

public void setUp() {

weatherApplication.start();

}

@After

public void tearDown() {

weatherApplication.stop();

}

@Test

public void servesWindSpeedBasedOnForecastIoResponse() throws IOException {

forecastIoService.stubFor(get(urlEqualTo("/forecast/e67b0e3784104669340c3cb089412b67/51.507253,-0.127755"))

.willReturn(aResponse().withBody("{\"currently\":{\"windSpeed\":12.34}}")));

Content content = Request.Get("http://localhost:" + weatherApplication.port() + "/wind-speed")

.execute()

.returnContent();

assertEquals("12.34mph", content.toString());

}

@Test

public void reportsErrorWhenForecastIoReturnsANonSuccessfulResponse() throws IOException {

forecastIoService.stubFor(get(urlEqualTo("/forecast/e67b0e3784104669340c3cb089412b67/51.507253,-0.127755"))

.willReturn(aResponse().withStatus(SC\_INTERNAL\_SERVER\_ERROR)));

HttpResponse httpResponse = Request.Get("http://localhost:" + weatherApplication.port() + "/wind-speed")

.execute()

.returnResponse();

assertEquals(503, httpResponse.getStatusLine().getStatusCode());

assertEquals("ERROR", IOUtils.toString(httpResponse.getEntity().getContent()));

}

}

This simple example is just a sampling of what the tool can do. Full sources can be found [on my github page](https://github.com/wojciechbulaty/examples/tree/master/wiremock-weaither-application). For more details see [Wiremock documentation](http://wiremock.org/docs/" \t "_blank).

**Manual exploratory testing**

We mentioned in our intro that there were two common use cases for over-the-wire stubs. The first was to act as a test double for automated tests, as we just saw. Next, comes a graphical console/web user interface for doing manual exploratory testing.

**Wiremock in the console**

Wiremock can be invoked from the command line and set up via an HTTP interface using for example [Postman](https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop) or cURL. You can set up a new stub by sending an HTTP post request to:  
http://<wiremock-host>:<wiremock-port>/\_\_admin/mappings/new

with a body:

{

"request": {

"method": "GET",

"url": "/some/thing"

},

"response": {

"status": 200,

"body": "Hello world!",

"headers": {

"Content-Type": "text/plain"

}

}

}

which is equivalent to executing the following Java code:

stubFor(get(urlEqualTo("/some/thing"))

.willReturn(aResponse()

.withHeader("Content-Type", "text/plain")

.withBody("Hello world!")));

**Manual testing issues**

If your testers are less tech-savvy or do not want to spend time dealing with technicalities, you can still help them concentrate on what they do best which is exploratory testing.

First, you could develop a web interface or a Swing GUI on top of Wiremock in-house.  In my experience it could take one to 10 developer days to perfect the solution over a period of up to 12 months.

Alternatively, you could use an off-the-shelf product such as [Traffic Parrot for Agile Teams](http://trafficparrot.com/#price). It is an over-the-wire test double with a web interface, to be used by developers and testers in agile teams. Developers can use Wiremock in their acceptance tests, and export stub definitions to Traffic Parrot as needed. Testers can then use the Traffic Parrot web UI to create test doubles for use in exploratory testing. I have personally had to go through that exercise several times at various companies so I consolidated the common solutions into Traffic Parrot for Agile Teams, which I have released to the public. It could save you from the frustration of going through the same mistakes I had made before I had developed it.

**Stubbing scenarios**

Another syndrome I have experienced during my consulting tenure is that there is a frequent requirement for more than one endpoint or more than one response stubbed for a given use case of the system under test. For example, if you want to present the current wind speed to the user based on their location you might have to talk to two web services, one to get the GPS coordinates and the second one to fetch the weather forecast. The JUnit test setup section test could look like this:

gpsService.stubFor(get(urlEqualTo("/what-are-my-coordinates"))

.willReturn(aResponse().withBody("12.34,-0.12")));

weatherService.stubFor(get(urlEqualTo("/forecast/12.34,-0.12"))

.willReturn(aResponse().withBody("{\"currently\":{\"windSpeed\":3.55}}")));

What the testers were looking for in this case was a drop-down in the web interface with pre-populated request/response pairs for the coordinates request, and the weather forecast coordinates request. That way, they would not have to look them up.

This functionality is available in Traffic Parrot as well, so you can use that to hit the ground running.

**Other tools**

If you would like to explore your options before committing to a particular tool there are plenty to choose from; I have found [more than 40](http://blog.trafficparrot.com/2016/08/service-virtualization-and-stubbing.html)alternatives to Wiremock that are coded in different programming languages and on different platforms providing varying sets of functionality.

Just to name a few alternatives:

* [Mountebank](http://www.mbtest.org/) (with bindings for Java, Python, C#, Clojure and many more)
* [VCR](https://github.com/vcr/vcr) (the most popular choice for Ruby and Ruby on Rails)
* [Stubby4j](https://github.com/vcr/vcr) (an alternative Java implementation with a different approach to designing the API)

Wiremock is still the most popular choice for Java today, with [more than 1400 stars and 40 contributors on Github](https://github.com/tomakehurst/wiremock).