WireMock A web service test double for all occasions

Fork me on GitHub

Stubbing

A core feature of WireMock is the ability to return canned HTTP responses for requests matching criteria. These criteria can be defined in terms of URL, headers and body content.

Basic stubbing

The following code will configure a response with a status of 200 to be returned when the relative URL exactly matches `/some/thing` (including query parameters). The body of the response will be “Hello world!” and a `Content-Type` header will be sent with a value of `text-plain`.

```
@Test
public void exactUrlOnly() {
    stubFor(get(urlEqualTo("/some/thing"))
            .willReturn(aResponse()
                .withHeader("Content-Type", "text/plai
                .withBody("Hello world!")));

    assertThat(testClient.get("/some/thing").statusCod
    assertThat(testClient.get("/some/thing/else").stat
}
```

Note

If you’d prefer to use slightly more BDDish language in your tests you can replace `stubFor` with `givenThat`.

To create the stub described above via the JSON API, the following document can either be posted to `http://<host>:<port>/__admin/mappings/new` or placed in a file with a `.json` extension under the `mappings` directory:

```
{
    "request": {
            "method": "GET",
            "url": "/some/thing"
    },
    "response": {
            "status": 200,
            "body": "Hello world!",
            "headers": {
                    "Content-Type": "text/plain"
            }
    }
}
```

HTTP methods currently supported are: `GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS`. You can specify `ANY` if you want the stub mapping to match on any request method.

URL matching

Entire URLs can be matched exactly (as in the example above) or via a regular expression. In Java this is done with the `urlMatching()` function:

```
stubFor(put(urlMatching("/thing/matching/[0-9]+"))
    .willReturn(aResponse().withStatus(200)));
```

And in JSON via the `urlPattern` attribute:

```
{
    "request": {
        "method": "PUT",
        "urlPattern": "/thing/matching/[0-9]+"
    },
    "response": {
        "status": 200
    }
}
```

Alternatively, just the path part of the URL can be matched exactly or using a regular expression, which is most useful when combined with query parameter matching (Query parameter matching):

```
stubFor(get(urlPathEqualTo("/query"))
    .willReturn(aResponse().withStatus(200)));
```

```
stubFor(get(urlPathMatching("/qu.*"))
    .willReturn(aResponse().withStatus(200)));
```

And in JSON via the `urlPath` attribute:

```
{
    "request": {
        "method": "GET",
        "urlPath": "/query"
    },
    "response": {
        "status": 200
    }
}
```

Request header matching

To match stubs according to request headers:

```
stubFor(post(urlEqualTo("/with/headers"))
    .withHeader("Content-Type", equalTo("text/xml"))
    .withHeader("Accept", matching("text/.*"))
    .withHeader("etag", notMatching("abcd.*"))
    .withHeader("X-Custom-Header", containing("2134"))
        .willReturn(aResponse().withStatus(200)));
```

Or

```
{
    "request": {
        "method": "POST",
        "url": "/with/headers",
        "headers": {
            "Content-Type": {
                "equalTo": "text/xml"
            },
            "Accept": {
                "matches": "text/.*"
            },
            "etag": {
                "doesNotMatch": "abcd.*"
            },
            "X-Custom-Header": {
                "contains": "2134"
            }
        }
    },
    "response": {
            "status": 200
    }
}
```

Query parameter matching

Query parameters can be matched in a similar fashion to headers:

```
stubFor(get(urlPathEqualTo("/with/query"))
    .withQueryParam("search", containing("Some text"))
        .willReturn(aResponse().withStatus(200)));
```

And in JSON:

```
{
    "request": {
        "method": "GET",
        "urlPath": "/with/query",
        "queryParameters": {
            "search": {
                "contains": "Some text"
            }
        }
    },
    "response": {
            "status": 200
    }
}
```

Note: you must use `urlPathEqualTo` or `urlPathMatching` to specify the path, as `urlEqualTo` or `urlMatching` will attempt to match the whole request URL, including the query parameters.

Request body matching

For PUT and POST requests the contents of the request body can be used to match stubs:

```
stubFor(post(urlEqualTo("/with/body"))
    .withRequestBody(matching("<status>OK</status>"))
    .withRequestBody(notMatching(".*ERROR.*"))
        .willReturn(aResponse().withStatus(200)));
```

Body content can be matched using all the same predicates as for headers: `equalTo`, `matching`, `notMatching`, `containing`.

The JSON equivalent of the above example would be:

```
{
    "request": {
        "method": "POST",
        "url": "/with/body",
        "bodyPatterns": [
            { "matches": "<status>OK</status>" },
            { "doesNotMatch": ".*ERROR.*" }
        ]
    },
    "response": {
            "status": 200
    }
}
```

JSON body matching

Body content which is valid JSON can be matched on semantically:

```
stubFor(post(urlEqualTo("/with/json/body"))
    .withRequestBody(equalToJson("{ \"houseNumber\": 4
    .willReturn(aResponse().withStatus(200)));
```

This uses JSONAssert internally. The default compare mode is `` `NON_EXTENSIBLE` `` by default, but this can be overridden:

```
.withRequestBody(equalToJson("{ \"houseNumber\": 4, \"
```

See JSONCompareMode for more details.

The JSON equivalent of the above example is:

```
{
    "request": {
        "method": "POST",
        "url": "/with/json/body",
        "bodyPatterns" : [
            { "equalToJson" : "{ \"houseNumber\": 4, \
        ]
    },
    "response": {
            "status": 200
    }
}
```

JSONPath expressions can also be used:

```
stubFor(post(urlEqualTo("/with/json/body"))
    .withRequestBody(matchingJsonPath("$.status"))
    .withRequestBody(matchingJsonPath("$.things[$(@.na
    .willReturn(aResponse().withStatus(201)));
```

The path syntax is implemented by the JSONPath library. A JSON body will be considered to match a path expression if the expression returns either a non-null single value (string, integer etc.), or a non-empty object or array.

The JSON equivalent of the above example would be:

```
{
    "request": {
        "method": "POST",
        "url": "/with/json/body",
        "bodyPatterns" : [
            { "matchesJsonPath" : "$.status"},
            { "matchesJsonPath" : "$.things[?(@.name =
        ]
    },
    "response": {
            "status": 201
    }
}
```

XML body matching

As with JSON, XML bodies can be matched on semantically.

In Java:

```
.withRequestBody(equalToXml("<thing>value</thing>"))
```

and in JSON:

```
"bodyPatterns" : [
    { "equalToXml" : "<thing>value</thing>" }
]
```

XPath body matching

Similar to matching on JSONPath, XPath can be used with XML bodies. An XML document will be considered to match if any elements are returned by the XPath evaluation.

```
stubFor(put(urlEqualTo("/xpath"))
    .withRequestBody(matchingXPath("/todo-list[count(t
    .willReturn(aResponse().withStatus(200)));
```

The JSON equivalent of which would be:

```
{
    "request": {
        "method": "PUT",
        "url": "/xpath",
        "bodyPatterns" : [
            { "matchesXPath" : "/todo-list[count(todo-
        ]
    },
    "response": {
            "status": 200
    }
}
```

To match XML with namespaced elements the namespaces must be registered:

```
stubFor(put(urlEqualTo("/namespaced/xpath"))
    .withRequestBody(matchingXPath("/stuff:outer/stuff
            .withXPathNamespace("stuff", "http://foo.c
    .willReturn(aResponse().withStatus(200)));
```

or:

```
{
    "request": {
        "method": "PUT",
        "url": "/xpath",
        "bodyPatterns" : [
            { "matchesXPath" : "/stuff:outer/stuff:inn
              "withXPathNamespaces" : {
                  "stuff" : "http://foo.com/"
              }
            },
        ]
    },
    "response": {
            "status": 200
    }
}
```

Note

All of the request matching options described here can also be used for Verifying.

Stub priority

It is sometimes the case that you'll want to declare two or more stub mappings that "overlap", in that a given request would be a match for more than one of them. By default, WireMock will use the most recently added matching stub to satisfy the request. However, in some cases it is useful to exert more control.

One example of this might be where you want to define a catch-all stub for any URL that doesn't match any more specific cases. Adding a priority to a stub mapping facilitates this:

```
//Catch-all case
stubFor(get(urlMatching("/api/.*")).atPriority(5)
    .willReturn(aResponse().withStatus(401)));

//Specific case
stubFor(get(urlEqualTo("/api/specific-resource")).atPr
    .willReturn(aResponse()
            .withStatus(200)
            .withBody("Resource state")));
```

Priority is set via the `priority` attribute in JSON:

```
{
    "priority": 1,
    "request": {
        "method": "GET",
        "url": "/api/specific-resource"
    },
    "response": {
        "status": 200
    }
}
```

Sending response headers

In addition to matching on request headers, it’s also possible to send response headers:

```
stubFor(get(urlEqualTo("/whatever"))
        .willReturn(aResponse()
                .withStatus(200)
                .withHeader("Content-Type", "applicati
                .withHeader("Cache-Control", "no-cache
```

Or

```
{
    "request": {
        "method": "GET",
        "url": "/whatever"
    },
    "response": {
        "status": 200,
        "headers": {
            "Content-Type": "text/plain",
            "Cache-Control": "no-cache"
        }
    }
}
```

Specifying the response body

The simplest way to specify a response body is as a string literal:

```
stubFor(get(urlEqualTo("/body"))
        .willReturn(aResponse()
                .withBody("Literal text to put in the
```

Or

```
{
    "request": {
        "method": "GET",
        "url": "/body"
    },
    "response": {
        "status": 200,
        "body": "Literal text to put in the body"
    }
}
```

To read the body content from a file, place the file under the `__files` directory. By default this is expected to be under `src/test/resources` when running from the JUnit rule. When running standalone it will be under the current directory in which the server was started. To make your stub use the file, simply call `bodyFile()` on the response builder with the file's path relative to `__files`:

```
stubFor(get(urlEqualTo("/body-file"))
        .willReturn(aResponse()
                .withBodyFile("path/to/myfile.xml")));
```

Or

```
{
    "request": {
        "method": "GET",
        "url": "/body-file"
    },
    "response": {
        "status": 200,
        "bodyFileName": "path/to/myfile.xml"
    }
}
```

Note

All strings used by WireMock, including the contents of body files are expected to be in `UTF-8` format. Passing strings in other character sets, whether by JVM configuration or body file encoding will most likely produce strange behaviour.

A response body in binary format can be specified as a `byte[]` via an overloaded `body()`:

```
stubFor(get(urlEqualTo("/binary-body"))
        .willReturn(aResponse()
                    .withBody(new byte[] { 1, 2, 3, 4 })))
```

The JSON API accepts this as a base64 string (to avoid stupidly long JSON documents):

```
{
    "request": {
        "method": "GET",
        "url": "/binary-body"
    },
    "response": {
        "status": 200,
        "base64Body" : "WUVTIElOREVFRCE="
    }
}
```

Saving stubs

Stub mappings which have been created can be persisted to the `mappings` directory via a call to `WireMock.saveAllMappings` in Java or posting a request with an empty body to `http://<host>:<port>/__admin/mappings/save`.

Note that this feature is not available when running WireMock from a servlet container.

Reset

The WireMock server can be reset at any time, removing all stub mappings and deleting the request log. If you're using either of the JUnit rules this will happen automatically at the start of every test case. However you can do it yourself via a call to `WireMock.reset()` in Java or posting a request with an empty body to `http://<host>:<port>/__admin/reset`.

If you've created some file based stub mappings to be loaded at startup and you don't want these to disappear when you do a reset you can call `WireMock.resetToDefault()` instead, or post an empty request to `http://<host>:<port>/__admin/mappings/reset`.