Behat API Extension Documentation

Release 2.3.1

Christer Edvartsen

Jan 29, 2020

Contents

An open source (MIT licensed) Behat extension that provides an easy way to test JSON-based APIs in Behat 3.

CHAPTER 1

Installation guide

1.1 Requirements

Behat API extension requires the following packages:

- PHP `^5.6`
- behat/behat `^3.0`
- guzzlehttp/guzzle `^6.0`
- beberlei/assert `^2.1`
- firebase/php-jwt `^4.0 | ^5.0`

You do not need to add any of these to your own `composer.json` file as the extension requires them.

1.2 Installation

1.2.1 Using composer

Install the extension by adding the following to your `composer.json` file:

```
{
  "require-dev": {
    "imbo/behat-api-extension": "^2.1"
  }
}
```

and then updating your dependencies by issuing `composer update imbo/behat-api-extension`.

1.3 Configuration

After you have installed the extension you need to activate it in your Behat configuration file (for instance `behat.yml`):

```
default:
    suites:
        default:
            # ...

    extensions:
        Imbo\BehatApiExtension: ~
```

The following configuration options are required for the extension to work as expected:

Key	Type	Default value	Description
`apiClient.base_uri`	string	http://localhost:8080	Base URI of the application under test. Must be connectable for the tests to execute.

It should be noted that everything in the `apiClient` configuration array is passed directly to the Guzzle Client instance used internally by the extension.

Example of a configuration file with several configuration entries:

```
default:
    suites:
        default:
            # ...

    extensions:
        Imbo\BehatApiExtension:
            apiClient:
                base_uri: http://localhost:8080
                timeout: 5.0
                verify: false
```

Refer to the Guzzle documentation for available configuration options for the Guzzle client.

1.4 Upgrading

This section will cover breaking changes between major versions and other related information to ease upgrading to the latest version.

1.4.1 Migrating from v1.x to v2.x

Changes

- *Configuration change*
- *Renamed public methods*
- *Updated steps*

- *Functions names for the JSON matcher*
- *Exceptions*

Configuration change

In `v1` the extension only had a single configuration option, which was `base_uri`. This is still an option in `v2`, but it has been added to an `apiClient` key.

v1 behat.yml

```
default:
    suites:
        default:
            # ...

    extensions:
        Imbo\BehatApiExtension:
            base_uri: http://localhost:8080
```

v2 behat.yml

```
default:
    suites:
        default:
            # ...

    extensions:
        Imbo\BehatApiExtension:
            apiClient:
                base_uri: http://localhost:8080
```

Renamed public methods

The following public methods in the `Imbo\BehatApiExtension\Context\ApiContext` class have been renamed:

v1 method name	v2 method name
givenIAttachAFileToTheRequest	addMultipartFileToRequest
givenIAuthenticateAs	setBasicAuth
givenTheRequestHeaderIs	addRequestHeader
giventhefollowingformparametersareset	setRequestFormParams
givenTheRequestBodyIs	setRequestBody
givenTheRequestBodyContains	setRequestBodyToFileResource
whenIRequestPath	requestPath
thenTheResponseCodeIs	assertResponseCodeIs
thenTheResponseCodeIsNot	assertResponseCodeIsNot
thenTheResponseReasonPhraseIs	assertResponseReasonPhraseIs
thenTheResponseStatusLineIs	assertResponseStatusLineIs
thenTheResponseIs	assertResponseIs
thenTheResponseIsNot	assertResponseIsNot
thenTheResponseHeaderExists	assertResponseHeaderExists
thenTheResponseHeaderDoesNotExist	assertResponseHeaderDoesNotExists
thenTheResponseHeaderIs	assertResponseHeaderIs
thenTheResponseHeaderMatches	assertResponseHeaderMatches
thenTheResponseBodyIsAnEmptyObject	assertResponseBodyIsAnEmptyJsonObject
thenTheResponseBodyIsAnEmptyArray	assertResponseBodyIsAnEmptyJsonArray
thenTheResponseBodyIsAnArrayOfLength	assertResponseBodyJsonArrayLength
thenTheResponseBodyIsAnArrayWithALengthOfAtLeast	assertResponseBodyJsonArrayMinLength
thenTheResponseBodyIsAnArrayWithALengthOfAtMost	assertResponseBodyJsonArrayMaxLength
thenTheResponseBodyIs	assertResponseBodyIs
thenTheResponseBodyMatches	assertResponseBodyMatches
thenTheResponseBodyContains	assertResponseBodyContainsJson

Some methods have also been removed (as the result of removed steps):

- whenIRequestPathWithBody
- whenIRequestPathWithJsonBody
- whenISendFile

Updated steps

v1 contained several When steps that could configure the request as well as sending it, in the same step. These steps has been removed in v2.0.0, and the extension now requires you to configure all aspects of the request using the Given steps prior to issuing one of the few When steps.

Removed / updated steps

- *Given the request body is :string*
- *When I request :path using HTTP :method with body: <PyStringNode>*
- *When I request :path using HTTP :method with JSON body: <PyStringNode>*
- *When I send :filePath (as :mimeType) to :path using HTTP :method*
- *Then the response body is an empty object*
- *Then the response body is an empty array*

- *Then the response body is an array of length* `:length`
- *Then the response body is an array with a length of at least* `:length`
- *Then the response body is an array with a length of at most* `:length`
- *Then the response body contains:* `<PyStringNode>`

Given the request body is `:string`

This step now uses a `<PyStringNode>` instead of a regular string:

v1

```
Given the request body is "some data"
```

v2

```
Given the request body is:
    """
    some data
    """
```

When I request `:path` using HTTP `:method` with body: `<PyStringNode>`

The body needs to be set using a `Given` step and not in the `When` step:

v1

```
When I request "/some/path" using HTTP POST with body:
    """
    {"some":"data"}
    """
```

v2

```
Given the request body is:
    """
    {"some":"data"}
    """
When I request "/some/path" using HTTP POST
```

When I request `:path` using HTTP `:method` with JSON body: `<PyStringNode>`

The `Content-Type` header and body needs to be set using `Given` steps:

v1

```
When I request "/some/path" using HTTP POST with JSON body:
    """
    {"some":"data"}
    """
```

v2

```
Given the request body is:
    """
    {"some":"data"}
    """
And the "Content-Type" request header is "application/json"
When I request "/some/path" using HTTP POST
```

When I send `:filePath` (as `:mimeType`) to `:path` using HTTP `:method`

These steps must be replaced with the following:

v1

```
When I send "/some/file.jpg" to "/some/endpoint" using HTTP POST
```

```
When I send "/some/file" as "application/json" to "/some/endpoint" using HTTP POST
```

v2

```
Given the request body contains "/some/file.jpg"
When I request "/some/endpoint" using HTTP POST
```

```
Given the request body contains "/some/file"
And the "Content-Type" request header is "application/json"
When I request "/some/endpoint" using HTTP POST
```

The first form in the old and new versions will guess the mime type of the file and set the `Content-Type` request header accordingly.

Then the response body is an empty object

Slight change that adds "JSON" in the step text for clarification:

v1

```
Then the response body is an empty object
```

v2

```
Then the response body is an empty JSON object
```

Then the response body is an empty array

Slight change that adds "JSON" in the step text for clarification:

v1

```
Then the response body is an empty array
```

v2

```
Then the response body is an empty JSON array
```

Then the response body is an array of length `:length`

Slight change that adds "JSON" in the step text for clarification:

v1

```
Then the response body is an array of length 5
```

v2

```
Then the response body is a JSON array of length 5
```

Then the response body is an array with a length of at least `:length`

Slight change that adds "JSON" in the step text for clarification:

v1

```
Then the response body is an array with a length of at least 5
```

v2

```
Then the response body is a JSON array with a length of at least 5
```

Then the response body is an array with a length of at most `:length`

Slight change that adds "JSON" in the step text for clarification:

v1

```
Then the response body is an array with a length of at most 5
```

v2

```
Then the response body is a JSON array with a length of at most 5
```

Then the response body contains: `<PyStringNode>`

Slight change that adds "JSON" in the step text for clarification:

v1

```
Then the response body contains:
    """
    {"some": "value"}
    """
```

v2

```
Then the response body contains JSON:
    """
    {"some": "value"}
    """
```

Functions names for the JSON matcher

When recursively checking a JSON response body, some custom functions exist that is represented as the value in a key / value pair. Below is a table of all available functions in `v1` along with the updated names used in `v2`:

`v1` function	`v2` function
`@length(num)`	`@arrayLength(num)`
`@atLeast(num)`	`@arrayMinLength(num)`
`@atMost(num)`	`@arrayMaxLength(num)`
`<re>/pattern/</re>`	`@regExp(/pattern/)`

`v2` have also added more such functions, refer to the *Custom matcher functions and targeting* section for a complete list.

Exceptions

The extension will from `v2` on throw native PHP exceptions or namespaced exceptions (like for instance `Imbo\BehatApiExtension\Exception\AssertionException`). In `v1` exceptions could come directly from `beberlei/assert`, which is the assertion library used in the extension. The fact that the extension uses this library is an implementation detail, and it should be possible to switch out this library without making any changes to the public API of the extension.

If versions after `v2` throws other exceptions it should be classified as a bug and fixed accordingly.

CHAPTER 2

End user guide

2.1 Set up the request

The following steps can be used prior to sending a request.

Available steps

- *Given I attach* `:path` *to the request as* `:partName`
- *Given the following multipart form parameters are set:* `<TableNode>`
- *Given I am authenticating as* `:username` *with password* `:password`
- *Given the* `:header` *request header is* `:value`
- *Given the* `:header` *request header contains* `:value`
- *Given the following form parameters are set:* `<TableNode>`
- *Given the request body is:* `<PyStringNode>`
- *Given the request body contains* `:path`
- *Given the response body contains a JWT identified by* `:name`*, signed with* `:secret:` `<PyStringNode>`

2.1.1 Given I attach `:path` to the request as `:partName`

Attach a file to the request (causing a `multipart/form-data` request, populating the `$_FILES` array on the server). Can be repeated to attach several files. If a specified file does not exist an `InvalidArgumentException` exception will be thrown. `:path` is relative to the working directory unless it's absolute.

Examples:

Step	:path	Entry in $_FILES on the server (:partName)
Given I attach "/path/to/file.jpg" to the request as "file1"	/path/to/file.jpg	$_FILES['file1']
Given I attach "c:\some\file.jpg" to the request as "file2"	c:\some\file.jpg	$_FILES['file2']
Given I attach "features/some.feature" to the request as "feature"	features/some.feature	$_FILES['feature']

This step can not be used when sending requests with a request body. Doing so results in an InvalidArgumentException exception.

2.1.2 Given the following multipart form parameters are set: <TableNode>

This step can be used to set form parameters (as if the request is a <form> being submitted). A table node must be used to specify which fields / values to send:

```
Given the following multipart form parameters are set:
    | name | value |
    | foo  | bar   |
    | bar  | foo   |
    | bar  | bar   |
```

The first row in the table must contain two values: name and value. The rows that follows are the fields / values you want to send. This step sets the HTTP method to POST by default and the Content-Type request header to multipart/form-data.

This step can not be used when sending requests with a request body. Doing so results in an InvalidArgumentException exception.

To use a different HTTP method, simply specify the wanted method in the *When I request :path using HTTP :method* step.

2.1.3 Given I am authenticating as :username with password :password

Use this step to set up basic authentication to the next request.

Examples:

Step	:username	:password
Given I am authenticating as "foo" with password "bar"	foo	bar

2.1.4 Given the :header request header is :value

Set the :header request header to :value. Can be repeated to set multiple headers. When repeated with the same :header the last value will be used.

Trying to force specific headers to have certain values combined with other steps that ends up modifying request headers (for instance attaching files) can lead to undefined behavior.

Examples:

Step	:header	:value
Given the "User-Agent" request header is "test/1.0"	User-Agent	test/1.0
Given the "Accept" request header is "application/json"	Accept	application/json

2.1.5 Given the :header request header contains :value

Add :value to the :header request header. Can be repeated to set multiple headers. When repeated with the same :header the header will be converted to an array.

Examples:

Step	:header	:value
Given the "X-Foo" request header contains "Bar"	X-Foo	Bar

2.1.6 Given the following form parameters are set: <TableNode>

This step can be used to set form parameters (as if the request is a <form> being submitted). A table node must be used to specify which fields / values to send:

```
Given the following form parameters are set:
    | name | value |
    | foo  | bar   |
    | bar  | foo   |
    | bar  | bar   |
```

The first row in the table must contain two values: name and value. The rows that follows are the fields / values you want to send. This step sets the HTTP method to POST by default and the Content-Type request header to application/x-www-form-urlencoded, unless the step is combined with *Given I attach :path to the request as :partName*, in which case the Content-Type request header will be set to multipart/form-data and all the specified fields will be sent as parts in the multipart request.

This step can not be used when sending requests with a request body. Doing so results in an InvalidArgumentException exception.

To use a different HTTP method, simply specify the wanted method in the *When I request :path using HTTP :method* step.

2.1.7 Given the request body is: <PyStringNode>

Set the request body to a string represented by the contents of the <PyStringNode>.

Examples:

```
Given the request body is:
    """
    {
        "some": "data"
    }
    """
```

2.1.8 Given the request body contains `:path`

This step can be used to set the contents of the file at `:path` in the request body. If the file does not exist or is not readable the step will fail.

Examples:

Step	`:path`
Given the request body contains "`/path/to/file`"	`/path/to/file`

The step will figure out the mime type of the file (using mime_content_type) and set the `Content-Type` request header as well. If you wish to override the mime type you can use the *Given the :header request header is :value* step **after** setting the request body.

2.1.9 Given the response body contains a JWT identified by `:name`, signed with `:secret: <PyStringNode>`

This step can be used to prepare the JWT custom matcher function with data that it is going to match on. If the response contains JWTs these can be registered with this step, then matched with the *Then the response body contains JSON: <PyStringNode>* step after the response has been received. The `<PyStringNode>` represents the payload of the JWT:

Examples:

```
Given the response body contains a JWT identified by "my JWT", signed with "some␣
↪secret":
    """
    {
        "some": "data",
        "value": "@regExp(/(some|expression)/i)"
    }
    """
```

The above step would register a JWT which can be matched with `@jwt(my JWT)` using the *@jwt()* custom matcher function. The way the payload is matched is similar to matching a JSON response body, as explained in the *Then the response body contains JSON: <PyStringNode>* section, which means *custom matcher functions* can be used, as seen in the example above.

2.2 Send the request

After setting up the request it can be sent to the server in a few different ways. Keep in mind that all configuration regarding the request must be done prior to any of the following steps, as they will actually send the request.

Available steps

- *When I request `:path`*
- *When I request `:path` using HTTP `:method`*

2.2.1 When I request `:path`

Request `:path` using HTTP GET. Shorthand for *When I request :path using HTTP GET*.

2.2.2 When I request `:path` using HTTP `:method`

`:path` is relative to the `base_uri` configuration option, and `:method` is any HTTP method, for instance `POST` or `DELETE`. If `:path` starts with a slash, it will be relative to the root of `base_uri`.

Examples:

Assume that the ‘‘base_uri‘‘ configuration option has been set to ‘‘http://example.com/dir‘‘ in the following examples.

Step	`:path`	`:method`	Resulting URI
When I request “`/?foo=bar&bar=foo`”	`/?foo=bar&bar=foo`	`GET`	`http://example.com/?foo=bar&bar=foo`
When I request “`/some/path`” using HTTP `DELETE`	`/some/path`	`DELETE`	`http://example.com/some/path`
When I request “`foobar`” using HTTP `POST`	`foobar`	`POST`	`http://example.com/dir/foobar`

2.3 Verify server response

After a request has been sent, some steps exist that can be used to verify the response from the server.

Available steps

- *Then the response code is `:code`*
- *Then the response code is not `:code`*
- *Then the response reason phrase is `:phrase`*
- *Then the response reason phrase is not `:phrase`*
- *Then the response reason phrase matches `:pattern`*
- *Then the response status line is `:line`*
- *Then the response status line is not `:line`*
- *Then the response status line matches `:pattern`*
- *Then the response is `:group`*
- *Then the response is not `:group`*
- *Then the `:header` response header exists*
- *Then the `:header` response header does not exist*
- *Then the `:header` response header is `:value`*
- *Then the `:header` response header is not `:value`*
- *Then the `:header` response header matches `:pattern`*
- *Then the response body is an empty JSON object*

2.3.1 Then the response code is `:code`

Asserts that the response code equals `:code`.

Examples:

- Then the response code is `200`
- Then the response code is `404`

2.3.2 Then the response code is not `:code`

Asserts that the response code **does not** equal `:code`.

Examples:

- Then the response code is not `200`
- Then the response code is not `404`

2.3.3 Then the response reason phrase is `:phrase`

Assert that the response reason phrase equals `:phrase`. The comparison is case sensitive.

Examples:

- Then the response reason phrase is "`OK`"
- Then the response reason phrase is "`Bad Request`"

2.3.4 Then the response reason phrase is not `:phrase`

Assert that the response reason phrase does not equal `:phrase`. The comparison is case sensitive.

Examples:

- Then the response reason phrase is not "`OK`"
- Then the response reason phrase is not "`Bad Request`"

2.3.5 Then the response reason phrase matches `:pattern`

Assert that the response reason phrase matches the regular expression `:pattern`. The pattern must be a valid regular expression, including delimiters, and can also include optional modifiers.

Examples:

- Then the response reason phrase matches "`/ok/i`"
- Then the response reason phrase matches "`/OK/`"

For more information regarding regular expressions and the usage of modifiers, refer to the PHP manual.

2.3.6 Then the response status line is `:line`

Assert that the response status line equals `:line`. The comparison is case sensitive.

Examples:

- Then the response status line is "`200 OK`"
- Then the response status line is "`304 Not Modified`"

2.3.7 Then the response status line is not `:line`

Assert that the response status line does not equal `:line`. The comparison is case sensitive.

Examples:

- Then the response status line is not "`200 OK`"
- Then the response status line is not "`304 Not Modified`"

2.3.8 Then the response status line matches `:pattern`

Assert that the response status line matches the regular expression `:pattern`. The pattern must be a valid regular expression, including delimiters, and can also include optional modifiers.

Examples:

- Then the response status line matches "`/200 ok/i`"
- Then the response status line matches "`/200 OK/`"

For more information regarding regular expressions and the usage of modifiers, refer to the PHP manual.

2.3.9 Then the response is `:group`

Asserts that the response is in `:group`.

Allowed groups and their response code ranges are:

Group	Response code range
`informational`	100 to 199
`success`	200 to 299
`redirection`	300 to 399
`client error`	400 to 499
`server error`	500 to 599

Examples:

- Then the response is "`informational`"
- Then the response is "`client error`"

2.3.10 Then the response is not `:group`

Assert that the response is not in `:group`.

Allowed groups and their ranges are:

Group	Response code range
`informational`	100 to 199
`success`	200 to 299
`redirection`	300 to 399
`client error`	400 to 499
`server error`	500 to 599

Examples:

- Then the response is not "`informational`"
- Then the response is not "`client error`"

2.3.11 Then the `:header` response header exists

Assert that the `:header` response header exists. The value of `:header` is case-insensitive.

Examples:

- Then the "`Vary`" response header exists
- Then the "`content-length`" response header exists

2.3.12 Then the `:header` response header does not exist

Assert that the `:header` response header does not exist. The value of `:header` is case-insensitive.

Examples:

- Then the "`Vary`" response header does not exist
- Then the "`content-length`" response header does not exist

2.3.13 Then the `:header` response header is `:value`

Assert that the value of the `:header` response header equals `:value`. The value of `:header` is case-insensitive, but the value of `:value` is not.

Examples:

- Then the "`Content-Length`" response header is "`15000`"
- Then the "`X-foo`" response header is "`foo, bar`"

2.3.14 Then the `:header` response header is not `:value`

Assert that the value of the `:header` response header **does not** equal `:value`. The value of `:header` is case-insensitive, but the value of `:value` is not.

Examples:

- Then the "`Content-Length`" response header is not "`15000`"
- Then the "`X-foo`" response header is not "`foo, bar`"

2.3.15 Then the `:header` response header matches `:pattern`

Assert that the value of the `:header` response header matches the regular expression `:pattern`. The pattern must be a valid regular expression, including delimiters, and can also include optional modifiers. The value of `:header` is case-insensitive.

Examples:

- Then the "`content-length`" response header matches "`/[0-9]+/`"
- Then the "`x-foo`" response header matches "`/(FOO|BAR)/i`"
- Then the "`X-FOO`" response header matches "`/^(foo|bar)$/`"

For more information regarding regular expressions and the usage of modifiers, refer to the PHP manual.

2.3.16 Then the response body is an empty JSON object

Assert that the response body is an empty JSON object (`{}`).

2.3.17 Then the response body is an empty JSON array

Assert that the response body is an empty JSON array (`[]`).

2.3.18 Then the response body is a JSON array of length `:length`

Assert that the length of the JSON array in the response body equals `:length`.

Examples:

- Then the response body is a JSON array of length `1`
- Then the response body is a JSON array of length `3`

If the response body does not contain a JSON array, the test will fail.

2.3.19 Then the response body is a JSON array with a length of at least `:length`

Assert that the length of the JSON array in the response body has a length of at least `:length`.

Examples:

- Then the response body is a JSON array with a length of at least `4`
- Then the response body is a JSON array with a length of at least `5`

If the response body does not contain a JSON array, the test will fail.

2.3.20 Then the response body is a JSON array with a length of at most `:length`

Assert that the length of the JSON array in the response body has a length of at most `:length`.

Examples:

- Then the response body is a JSON array with a length of at most `4`
- Then the response body is a JSON array with a length of at most `5`

If the response body does not contain a JSON array, the test will fail.

2.3.21 Then the response body is: `<PyStringNode>`

Assert that the response body equals the text found in the `<PyStringNode>`. The comparison is case-sensitive.

Examples:

```
Then the response body is:
    """
    {"foo":"bar"}
    """
```

```
Then the response body is:
    """
    foo
    """
```

2.3.22 Then the response body is not: `<PyStringNode>`

Assert that the response body **does not** equal the value found in `<PyStringNode>`. The comparison is case sensitive.

Examples:

```
Then the response body is not:
    """
    some value
    """
```

2.3.23 Then the response body matches: `<PyStringNode>`

Assert that the response body matches the regular expression pattern found in `<PyStringNode>`. The expression must be a valid regular expression, including delimiters and optional modifiers.

Examples:

```
Then the response body matches:
    """
    /^{"FOO": ?"BAR"}$/i
    """
```

```
Then the response body matches:
    """
    /foo/
    """
```

2.3.24 Then the response body contains JSON: <PyStringNode>

Used to recursively match the response body (or a subset of the response body) against a JSON blob.

In addition to regular value matching some custom matching-functions also exist, for asserting value types, array lengths and so forth. There is also a regular expression type matcher that can be used to match string values.

Regular value matching

Assume the following JSON response for the examples in this section:

```
{
  "string": "string value",
  "integer": 123,
  "double": 1.23,
  "boolean": true,
  "null": null,
  "object":
  {
    "string": "string value",
    "integer": 123,
    "double": 1.23,
    "boolean": true,
    "null": null,
    "object":
    {
      "string": "string value",
      "integer": 123,
      "double": 1.23,
      "boolean": true,
      "null": null
    }
  },
  "array":
  [
    "string value",
    123,
    1.23,
    true,
    null,
    {
      "string": "string value",
      "integer": 123,
      "double": 1.23,
      "boolean": true,
      "null": null
    }
  ]
}
```

Example: Regular value matching of a subset of the response

```
Then the response body contains JSON:
    """
    {
      "string": "string value",
      "boolean": true
    }
    """
```

Example: Check values in objects

```
Then the response body contains JSON:
    """
    {
      "object":
      {
        "string": "string value",
        "object":
        {
          "null": null,
          "integer": 123
        }
      }
    }
    """
```

Example: Check numerically indexed array contents

```
Then the response body contains JSON:
    """
    {
      "array":
      [
        true,
        "string value",
        {
          "integer": 123
        }
      ]
    }
    """
```

Notice that the order of the values in the arrays does not matter. To be able to target specific indexes in an array a special syntax needs to be used. Please refer to *Custom matcher functions and targeting* for more information and examples.

Custom matcher functions and targeting

In some cases the need for more advanced matching arises. All custom functions is used in place of the string value they are validating, and because of the way JSON works, they need to be specified as strings to keep the JSON valid.

- *Array length* - `@arrayLength` / `@arrayMaxLength` / `@arrayMinLength`
- *Variable type* - `@variableType`
- *Regular expression matching* - `@regExp`

- *Match specific keys in a numerically indexed array - `<key>[<index>]`*
- *Numeric comparison - `@gt` / `@lt`*
- *JWT token matching - `@jwt`*

Array length - `@arrayLength` / `@arrayMaxLength` / `@arrayMinLength`

Three functions exist for asserting the length of regular numerically indexed JSON arrays, `@arrayLength`, `@arrayMaxLength` and `@arrayMinLength`. Given the following response body:

```
{
  "items":
  [
    "foo",
    "bar",
    "foobar",
    "barfoo",
    123
  ]
}
```

one can assert the exact length using `@arrayLength`:

```
Then the response body contains JSON:
    """
    {"items": "@arrayLength(5)"}
    """
```

or use the relative length matchers:

```
Then the response body contains JSON:
    """
    {"items": "@arrayMaxLength(10)"}
    """
And the response body contains JSON:
    """
    {"items": "@arrayMinLength(3)"}
    """
```

Variable type - `@variableType`

To be able to assert the variable type of specific values, the `@variableType` function can be used. The following types can be asserted:

- `boolean` / `bool`
- `integer` / `int`
- `double` / `float`
- `string`
- `array`
- `object`

- `null`
- `scalar`
- `any`

Given the following response:

```
{
  "boolean value": true,
  "int value": 123,
  "double value": 1.23,
  "string value": "some string",
  "array value": [1, 2, 3],
  "object value": {"foo": "bar"},
  "null value": null,
  "scalar value": 3.1416
}
```

the type of the values can be asserted like this:

```
Then the response body contains JSON:
    """
    {
      "boolean value": "@variableType(boolean)",
      "int value": "@variableType(integer)",
      "double value": "@variableType(double)",
      "string value": "@variableType(string)",
      "array value": "@variableType(array)",
      "object value": "@variableType(object)",
      "null value": "@variableType(null)",
      "scalar value": "@variableType(scalar)"
    }
    """
```

The `boolean`, `integer` and `double` types can also be expressed using `bool`, `int` and `float` respectively. There is no difference in the actual validation being executed.

For the `@variableType(scalar)` assertion refer to the is_scalar function in the PHP manual as to what is considered to be a scalar.

When using `any` as a type, the validation will basically allow any types, including `null`. One can also match against multiple types using `|` (for instance `@variableType(int|double|string)`). When using multiple types the validation will succeed (and stop) as soon as the value being tested matches one of the supplied types. Validation is done in the order specified.

Regular expression matching - `@regExp`

To use regular expressions to match values, the `@regExp` function exists, that takes a regular expression as an argument, complete with delimiters and optional modifiers. Example:

```
Then the response body contains JSON:
    """
    {
      "foo": "@regExp(/(some|expression)/i)",
      "bar":
      {
        "baz": "@regExp(/[0-9]+/)"
```

(continues on next page)

(continued from previous page)

```
      }
    }
    """
```

This can be used to match variables of type `string`, `integer` and `float`/`double` only, and the value that is matched will be cast to a string before doing the match. Refer to the PHP manual regarding how regular expressions work in PHP.

Match specific keys in a numerically indexed array - `<key>[<index>]`

If you need to verify an element at a specific index within a numerically indexed array, use the `key[<index>]` notation as the key, and not the regular field name. Consider the following response body:

```
{
  "items":
  [
    "foo",
    "bar",
    {
      "some":
      {
        "nested": "object",
        "foo": "bar"
      }
    },
    [1, 2, 3]
  ]
}
```

If you need to verify the values, use something like the following step:

```
Then the response body contains JSON:
    """
    {
      "items[0]": "foo",
      "items[1]": "@regExp(/(foo|bar|baz)/)",
      "items[2]":
      {
        "some":
        {
          "foo": "@regExp(/ba(r|z)/)"
        }
      },
      "items[3]": "@arrayLength(3)"
    }
    """
```

If the response body contains a numerical array as the root node, you will need to use a special syntax for validation. Consider the following response body:

```
[
  "foo",
  123,
  {
    "foo": "bar"
```

(continues on next page)

(continued from previous page)

```
  },
  "bar",
  [1, 2, 3]
]
```

To validate this, use the following step:

```
Then the response body contains JSON:
    """
    {
      "[0]": "foo",
      "[1]": 123,
      "[2]":
      {
        "foo": "bar"
      },
      "[3]": "@regExp(/bar/)",
      "[4]": "@arrayLength(3)"
    }
    """
```

Numeric comparison - `@gt` / `@lt`

To verify that a numeric value is greater than or less than a value, the `@gt` and `@lt` functions can be used respectively. Given the following response body:

```
{
  "some-int": 123,
  "some-double": 1.23,
  "some-string": "123"
}
```

one can compare the numeric values using:

```
Then the response body contains JSON:
    """
    {
      "some-int": "@gt(120)",
      "some-double": "@gt(1.20)",
      "some-string": "@gt(120)"
    }
    """
And the response body contains JSON:
    """
    {
      "some-int": "@lt(125)",
      "some-double": "@lt(1.25)",
      "some-string": "@lt(125)"
    }
    """
```

JWT token matching - `@jwt`

To verify a JWT in the response body the `@jwt()` custom matcher function can be used. The argument it takes is the name of a JWT token registered with the *Given the response body contains a JWT identified by :name, signed with :secret: <PyStringNode>* step earlier in the scenario.

Given the following response body:

```
{
  "value": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiU29tZSB1c2VyIn0.DsGGNmDD-
↪PBnwMLiQxeSHDGmKBSdP0lSmWuaiwSxfQE"
}
```

one can validate the JWT using a combination of two steps:

```
# Register the JWT
Given the response body contains a JWT identified by "my JWT", signed with "secret":
    """
    {
        "user": "Some user"
    }
    """

# Other steps ...

# After the request has been made, one can match the JWT in the response
And the response body contains JSON:
    """
    {
      "value": "@jwt(my JWT)"
    }
    """
```

2.4 Extending the extension

If you want to implement your own assertions, or for instance add custom authentication for all requests made against your APIs you can extend the context class provided by the extension to access the client, request, request options, response and the array contains comparator properties. These properties are accessed via the protected `$this->client`, `$this->request`, `$this->requestOptions`, `$this->response` and `$this->arrayContainsComparator` properties respectively. Keep in mind that `$this->response` is not populated until the client has made a request, i.e. after any of the aforementioned `@When` steps have finished.

2.4.1 Add `@Given`'s, `@When`'s and/or `@Then`'s

If you want to add a `@Given`, `@When` and/or `@Then` step, simply add a method in your `FeatureContext` class along with the step using annotations in the `phpdoc` block:

```
<?php
use Imbo\BehatApiExtension\Context\ApiContext;
use Imbo\BehatApiExtension\Exception\AssertionFailedException as Failure;

class FeatureContext extends ApiContext {
    /**
```

(continues on next page)

(continued from previous page)

```
     * @Then I want to check something
     */
    public function assertSomething() {
        // do some assertions on $this->response, and throw a Failure exception is the
        // assertion fails.
    }
}
```

With the above example you can now use `Then I want to check something` can be used in your feature files along with the steps defined by the extension.

2.4.2 Manipulate the API client

If you wish to manipulate the API client (`GuzzleHttp\Client`) this can be done in the initialization-phase:

```
<?php
use Imbo\BehatApiExtension\Context\ApiContext;
use GuzzleHttp\ClientInterface;
use GuzzleHttp\Middleware;
use Psr\Http\Message\RequestInterface;

class FeatureContext extends ApiContext {
    /**
     * Manipulate the API client
     *
     * @param ClientInterface $client
     * @return self
     */
    public function setClient(ClientInterface $client) {
        $stack = $client->getConfig('handler');
        $stack->push(Middleware::mapRequest(function(RequestInterface $request) {
            // Add something to the request and return the new instance
            return $request->withAddedHeader('Some-Custom-Header', 'some value');
        }));

        return parent::setClient($client);
    }
}
```

2.4.3 Register custom matcher functions

The extension comes with some built in matcher functions used to verify JSON-content (see *Then the response body contains JSON: <PyStringNode>*), like for instance `@arrayLength` and `@regExp`. These functions are basically callbacks to PHP methods / functions, so you can easily define your own and use them in your tests:

```
<?php
use Imbo\BehatApiExtension\Context\ApiContext;
use Imbo\BehatApiExtension\ArrayContainsComparator;

class FeatureContext extends ApiContext {
    /**
     * Add a custom function called @gt to the comparator
     *
```

(continues on next page)

(continued from previous page)

```
     * @param ArrayContainsComparator $comparator
     * @return self
     */
    public function setArrayContainsComparator(ArrayContainsComparator $comparator) {
        $comparator->addFunction('gt', function($num, $gt) {
            $num = (int) $num;
            $gt = (int) $gt;

            if ($num <= $gt) {
                throw new InvalidArgumentException(sprintf(
                    'Expected number to be greater than %d, got: %d.',
                    $gt,
                    $num
                ));
            }
        });

        return parent::setArrayContainsComparator($comparator);
    }
}
```

The above snippet adds a custom matcher function called `@gt` that can be used to check if a number is greater than another number. Given the following response body:

```
{
  "number": 42
}
```

the number in the `number` key could be verified with:

```
Then the response body contains JSON:
    """
    {
      "number": "@gt(40)"
    }
    """
```