Mockservr Documentation

Release latest

Jul 16, 2020

Contents

Mockservr is an API mocking system, allowing to configure endpoints, specifying requests details and responses.

- YAML or JSON endpoint configuration
- Manage endpoint configuration with API
- Ability to use Apache Velocity Template files to dynamically adapt responses to requests parameters
- Available under docker image, see dockerhub to be easily integrated into a development / test stack

Using Mockservr gives the ability to avoid real API calls to external providers, while ensuring that the code carries no logic related to the environment.

CHAPTER 1

Summary

1.1 Quickstart

Mockservr is designed to be as user-friendly as possible, hence it can be integrated into an existing stack really quickly, immediately providing a way to mock APIs.

1.1.1 System requirements

In order to run mockservr in the easiest way, we provide docker images.

The minimum requirements on the host system are:

- Docker >= 18 (https://www.docker.com/)

1.1.2 Running with docker

One way of running Mockservr is through Docker.

```
docker run -p 8080:80 -p 4580:4580 -v /mocks-directory:/usr/src/app/mocks rvip/
↪mockservr
```

From now, all defined mocks will be accessible through *http://localhost:8088*.

1.1.3 Running with docker-compose

Assuming your stack runs using docker-compose, Mockservr can be easily integrated in your docker-compose.yaml file :

```
services:
  # ...

  mockservr:
    image: rvip/mockservr
    volumes:
      - ./mocks-directory:/usr/src/app/mocks
    ports:
      - 8088:80
      - 4580:4580
```

From now on, all defined mocks will be accessible through *http://localhost:8088*.

1.2 HTTP Mocking

Mockservr allows you to mock HTTP endpoints, defined through YAML/JSON files. Usin HTTP Mocking, you can easily and rapidely mock any HTTP based API.

1.2.1 Overview

As for any mock in Mockservr, it must be defined in a YAML/JSON file, with the *.mock.yaml* or *mock.yml* or *mock.json* file extension.

HTTP endpoints are defined under a single object, named *http*, which should contain an array of endpoints.

Accessing the HTTP endpoints

Once Mockservr is up and running as described in *Quickstart*, it is possible to access the endpoints through your localhost and the port *8080*.

Note: The port on which Mockservr can be reached through HTTP can be customized, either by using the *-p* option if running with Docker (see *Running with docker*) or by using the *services.XX.port* option if running under docker-compose (see *Running with docker-compose*).

Defining the HTTP endpoints

There are two ways to define the HTTP endpoints in the Mockservr.

- Describe endpoints in an array under *http* in a YAML/JSON file
- Use *API*

Example of endpoints mock file

Using an array of objects under *http* will define endpoints for the mock:

Listing 1: YAML

```yaml
http:
  -
    request:
      path: '/foo'
    response:
      body: 'Hello World!'
  -
    request:
      path: '/bar'
    response:
      body: 'Hello bar!'
```

Listing 2: JSON

```json
{
  "http": [
    {
      "request": {
        "path": "/foo"
      },
      "response": {
        "body": "Hello World!"
      }
    },
    {
      "request": {
        "path": "/bar"
      },
      "response": {
        "body": "Hello bar!"
      }
    }
  ]
}
```

These endpoints are then accessible through HTTP:

```
curl -XGET 'http://localhost:8080/foo'
curl -XGET 'http://localhost:8080/bar'
```

Note: It is possible to define a single endpoint for a mock, by using an object instead of an array under *http*.

1.2.2 Endpoint

An Endpoint is defined as one or more responses that correspond to one or more requests, all of them defined under *http*. If *http* is an array, then each element is an Endpoint. If *http* is an object, then the object is the only endpoint of the mock.

An Endpoint defines at least a *request* and an *response* property.

Mandatory properties

request option

More details about the *request* option are availble in *Request*.

response option

More details about the *response* option are available in *Response*.

Endpoint Options

crossOrigin option

The *crossOrigin* option enables cross origin requests on Mockservr. This value can either be a boolean or an object.

Listing 3: YAML

```yaml
http:
  crossOrigin: true
  request:
    path: '/foo'
  response:
    body: 'Hello World!'
```

Listing 4: JSON

```json
{
  "http": {
    "crossOrigin": true
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

If a boolean is given, it will authorize HTTP *OPTION* requests on the endpoint.

The default response to the *OPTION* method is the following JSON:

Listing 5: JSON

```
  {
    "headers": {
      "access-control-allow-credentials": true,
      "access-control-allow-headers": "request.headers['access-control-request-headers
↪'] || '*'",
      "access-control-allow-methods": "GET,HEAD,POST,PUT,DELETE,CONNECT,OPTIONS,TRACE,
↪PATCH",
      "access-control-allow-origin": "*",
      "access-control-max-age": 3600
```

(continues on next page)

(continued from previous page)

```
    },
    "body": ""
  }
```

Note: The value of *access-control-allow-headers* is either equal to the request's *access-control-request-headers* header if defined or * (allowing all headers).

If *crossOrigin* option is an object, it must be a Response object (see *Response* for more information about Response's options). It overrides the default response as defined above.

Note: All headers defined within the *crossOrigin* options will be present in the response sent by Mockservr to any incoming HTTP request that matches the endpoint. These headers can be overwritten using the *headers* option of the *response* object.

maxCalls option

The *maxCalls* option defines a maximum calls count for a given Endpoint. It does not provide validator inference, as the only possible value is a plain integer.

Listing 6: YAML

```
http:
  maxCalls: 5
  request:
    path: '/foo'
  response:
    body: 'Hello World!'
```

Listing 7: JSON

```
{
  "http": {
    "maxCalls": 5
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

In the above example, the Endpoint is going to be positively matched 5 times. When the 6th call arrives, Mockservr will not match it positively against this Endpoint.

rateLimit option

The *rateLimit* option may either be a plain integer or an object. If a plain integer, it is the maximum number of accepted calls on this endpoint within one second. If an object, it must define a *callCount* property which is a plain

integer defining the number of accepted calls and a *interval* property defining the time window in milliseconds in which the *callCount* lies.

Listing 8: YAML

```yaml
http:
  rateLimit:
    callCount: 2
    interval: 5000
  request: '/foo'
  response: 'Hello World!'
```

Listing 9: JSON

```json
{
  "http": {
    "rateLimit": {
      "callCount": 2,
      "interval": 5000
    },
    "request": "/foo",
    "response": "Hello World!"
  }
}
```

In the above example, the API has a rate limit of 2 calls every 5 seconds.

Note: It is possible to add a custom Response in the *html.rateLimit* object.

1.2.3 Request

This section covers the *http.request* part of the endpoint definition ; it defines how Mockservr will match the incoming HTTP requests, and what response it will serve to the client.

Request Definition

The *http.request* may either be a string, an object or an array.

Mockservr comes with a **inference** feature, which means, if you do not explicitly define full *request* options to use, Mockservr will guess it for you.

Basic definition of a Request

The simpliest way to define a Request is by only defining its path. Mockservr allows you to write this path directly under *http.request*, using a string, such as:

Listing 10: YAML

```yaml
http:
  request: '/foo'
  response:
    body: 'Hello World!"
```

Listing 11: JSON

```
{
  "http": {
    "request": "/foo",
    "response": {
      "body": "Hello World!"
    }
  }
}
```

The endpoint is then accessible through HTTP:

```
curl -XGET 'http://localhost:8080/foo'
```

Note: This way to define an endpoint is equal to:

Listing 12: YAML

```
http:
  request:
    -
     path: '/foo'
  response:
    body: 'Hello World!'
```

Listing 13: JSON

```
{
  "http": {
    "request": [
      {
        "path": "/foo"
      }
    ],
    "response": {
      "body": "Hello World!"
    }
  }
}
```

Defining a single Request

If your endpoint must react to a single type of Request, then you can use an object to define it. To learn about all possible options to define the Request, please see ‘Requests Options‘http_mocking_request_options_.

Listing 14: YAML

```
http:
  request:
    path: '/foo'
  response:
    body: 'Hello World!'
```

Listing 15: JSON

```
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

The endpoint is then accessible through HTTP:

```
curl -XGET 'http://localhost:8080/foo'
```

Note: This way to define an endpoint is equal to:

Listing 16: YAML

```
http:
  request:
    -
     path: '/foo'
  response:
    body: 'Hello World!'
```

Listing 17: JSON

```
{
  "http": {
    "request": [
      {
        "path": "/foo"
      }
    ],
    "response": {
      "body": "Hello World!"
    }
  }
}
```

Defining multiple Request

In case your endpoint should serve a similar Response to requests that may have different shapes, you can define multiple matching Requests for the endpoint, by using an array.

Listing 18: YAML

```
http:
  request:
```

(continues on next page)

(continued from previous page)

```
    -
      path: '/foo'
    -
      path: '/bar'
  response:
    body: 'Hello World!'
```

Listing 19: JSON

```
{
  "http": {
    "request": [
      {
        "path": "/foo"
      },
      {
        "path": "/bar"
      }
    ],
    "response": {
      "body": "Hello World!"
    }
  }
}
```

The endpoint is then available through different HTTP requests:

```
curl -XGET 'http://localhost:8080/foo'
curl -XGET 'http://localhost:8080/bar'
```

Request Options

This section describes all the options available for a Request. For an incoming HTTP request to match a defined Request, it must match positively against all options.

The options are defined under the *request* object of an HTTP endpoint.

Most option can be described as a *Validators*, but may also be described as a scalar value, for which Mockservr will perform validator inference.

Defining multiple sets of options for a single Request

It is possible to describe multiple sets of options to describe a Request. To do so, the *request* must be an array of objects instead of a single object.

For each incoming HTTP request, Mockservr will try to match against all different Requests that have been defined.

It allows you to describe several ways to reach a single endpoint.

Listing 20: YAML

```
  http:
    request:
      -
```

(continues on next page)

(continued from previous page)

```
      path: '/foo'
    -
      path: '/bar'
  response:
    body: 'Hello World!'
```

Listing 21: JSON

```
{
  "http": {
    "request": [
      {
        "path": "/foo"
      },
      {
        "path": "/bar"
      }
    ]
    "response": {
      "body": "Hello World!"
    }
  }
}
```

Then, the two following HTTP requests will lead to the same response:

```
curl -XGET 'http://localhost:8080/foo'
curl -XGET 'http://localhost:8080/bar'
```

path option (required)

The *path* option defines on which path the endpoint can be reached. The *path* option matching is powered by path-to-regexp.

Listing 22: YAML

```
http:
  request:
    path: '/foo/:id'
  response:
    body: 'Hello World!'
```

Listing 23: JSON

```
{
  "http": {
    "request": {
      "path": "/foo/:id",
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

The example above is an endpoint reachable on the */foo/[id]* path of Mockservr (typically *http://localhost:8080*)

```
curl -XGET 'http://localhost:8080/foo/1'
```

basepath option

The *basepath* option allows to define the base path of the request. It is mainly useful to group requests by their base path in the Mockservr GUI. *basepath* will concat with *path* option before matching

Listing 24: YAML

```yaml
  http:
    request:
      basepath: '/foo'
      path: '/:id'
    response:
      body: 'Hello World!'
```

Listing 25: JSON

```
{
  "http": {
    "request": {
      "basepath": "/foo",
      "path": "/:id"
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

```
curl -XGET 'http://localhost:8080/foo/1'
```

body option

The *body* option allows you to define what the incoming HTTP request's body must look like. For it to be working as an object, the *Content-Type* header must be defined as *application/x-www-form-urlencoded* or as *application/json*. If not it will be working as string.

All types of *Validators* may be used with the *body* option. In case the body is a JSON or a form, it is possible to use an object under *body*. If not defined *body* will match any incoming request.

Listing 26: YAML

```yaml
  http:
    request:
      path: '/foo'
      body:
        name: 'John'
        last_name: ['Doe', 'Bar']
    response:
      body: 'Hello World!'
```

Listing 27: JSON

```
{
  "http": {
    "request": {
      "path": "/foo",
      "body": {
        "name": "John",
        "last_name": ["Doe", "Bar"]
      }
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

In the above example, the JSON or form body must define two key/value pairs: The first one is *name* and its value must be "John" (the *equal* validator is automatically inferred) ; the second one is *last_name* and its value must either be "Doe" or "Bar" (the *anyOf* validator is automatically inferred).

The incoming request's body may also be a simple string or any other scalar then the *equal* validator is automatically inferred.

Listing 28: YAML

```
http:
  request:
    path: '/foo'
    body: "Hello"
  response:
    body: 'Hello World!'
```

Listing 29: JSON

```
{
  "http": {
    "request": {
      "path": "/foo",
      "body": "Hello"
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

headers option

The *headers* options allows the control of the incoming HTTP request. **It must be an object** be an object with key/value pairs. The key is the header's name, and the value is the expected value.

Each object's property value can be any type of *Validators*, allowing a fine-grain control of the headers. If not defined *headers*, will match any incoming request.

Listing 30: YAML

```yaml
http:
  request:
    path: '/foo'
    headers:
      Content-Type: ['application/json', 'application/x-www-form-urlencoded']
  response:
    body: 'Hello World!'
```

Listing 31: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo",
      "headers": ['application/json', 'application/x-www-form-urlencoded']
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

In the above example, the endpoint will be triggered in the incoming HTTP request contains a *Content-Type* header and if its value is either *application/json* or *application/x-www-form-urlencoded* (the *anyOf* validator is automatically inferred).

method option

The *method* option defines which type of HTTP requests will match positively with the endpoint. Apart of the usual HTTP verbs (GET, POST, . . .), it is possible to set custom HTTP verbs.

Each object's property value can be any type of *Validators*, allowing a fine-grain control of the incoming query parameters. If not defined, *query* will match any incoming request.

Listing 32: YAML

```yaml
http:
  request:
    path: '/foo'
    method: ['GET', 'POST']
  response:
    body: 'Hello World!'
```

Listing 33: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo",
      "method": ["GET", "POST"]
    },
    "response": {
      "body": "Hello World!"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

The Request defined above will match positively against any incoming HTTP request which is a GET or a POST request (the *anyOf* validator is automatically inferred).

query option

The *query* option defines how the incoming HTTP request's query parameters will be matched. **It must be an object** in which each key/value pair correspond to a query parameter/value.

The value in each key/value pair is a *Validators*. If not defined *query* will match any incoming request.

Listing 34: YAML

```yaml
http:
  request:
    path: '/foo'
    query:
      action: 'show'
      id:
        type: 'typeOf'
        Value: 'number'
  response:
    body: 'Hello World!'
```

Listing 35: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo",
      "query": {
        "action": "show",
        "id": {
          "type": "typeOf",
          "value": "number"
        }
      }
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

To match the above Request, the incoming HTTP request must be of the form */foo?action=show&id=3*

1.2.4 Response

When Mockservr matches an incoming HTTP request to a Request defined in an endpoint, it will send back a HTTP response. The definition of this response lies into the *http.response* object.

It is possible to define several Responses for an endpoint and the way Mockservr should pick one of Responses from the incoming HTTP request.

Response Definition

A Response is defined by a set of options that describe Mockservr how to build the HTTP response to an incoming HTTP request that has been match successfully to the endpoint.

It is also possible to define several possible Responses for a single endpoint. To do so, *http.response* must be an array of objects, each object defining a possible Response. In that case, each object must also contains some options that will tell Mockservr how to pick the right Response ; otherwise, Mockservr will pick one of Response in the array.

Defining a single Response

Basically, an HTTP response is composed of a body and a status code. By default, the status code returned by Mockservr is 200 if not defined otherwise.

A basic Response definition would be:

Listing 36: YAML

```yaml
http:
  request:
    path: '/foo'
  response:
    body: 'Hello World!'
```

Listing 37: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

This endpoint definition will match any incoming request on */foo*, and the HTTP response's body will be "Hello World!" with HTTP status code 200.

Defining several possible Responses

Using an array instead of an object under *http.response*, it is possible to define several possible Response for a single endpoint. It is then possible to give a weight to each of the Responses, and Mockservr will pick one of the Response randomly, according to their respective weight.

Listing 38: YAML

```yaml
http:
  request:
    path: '/foo'
```

(continues on next page)

(continued from previous page)

```yaml
  response:
    -
      body: 'Hello World!'
      weight: 1
    -
      body: "Bye World!"
      weight: 1
```

Listing 39: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": [
      {
        "body": "Hello World!",
        "weight": 1
      },
      {
        "body": "Bye World!",
        weight: 1
      }
    ]
  }
}
```

For the endpoint defined above, Mockservr will pick a random Response ; as both their weights are 1, they'll be pick randomly with equal chances. See *http_mocking_response_weight_option*.

Response Options

Response options let you specify what content Mockservr will send as a response to a matching incoming HTTP request.

body option

The *body* option lets you specify the body of the response. An object is expected but it can be a string.

Body object must have two attributes *type* and *value*. *type* must be *plaintext* or *file*. The *plaintext* type lets you specify the response to send. The *value* attribute is the response body content. The *file* type lets you specify the path to a file that contain the response to send. The file may be of any type, which lets you define JSON responses, XML responses, pictures, … The *value* attribute is the path where to fetch the file, relatively to the mock file.

The incoming response body may also be a simple string then a *plaintext* type is automatically inferred.

Listing 40: YAML

```yaml
http:
  request:
    path: '/foo'
  response:
    body: 'Hello World!'
```

Listing 41: JSON

```
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

Listing 42: YAML

```
http:
  request:
    path: '/foo'
  response:
    body:
      type: 'plaintext'
      value: 'Hello World!'
```

Listing 43: JSON

```
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": {
        "type": "plaintext",
        "value": "Hello World!"
      }
    }
  }
}
```

In the previous examples, the response body will be *Hello World!*.

Listing 44: YAML

```
http:
  request:
    path: '/foo'
  response:
    body:
      type: 'file'
      value: './responses/foo/response.json'
```

Listing 45: JSON

```
{
  "http": {
    "request": {
```

(continues on next page)

(continued from previous page)

```
        "path": "/foo"
      },
      "response": {
        "body": {
          "type": "file",
          "value": "./responses/foo/response.json"
        }
      }
    }
  }
```

In the previous examples, the response file must be located in a *responses/foo/* directory from the mock's directory, within a *response.json* file.

Note: For pictures, only the following mime types are allowed: *image/gif*, *image/jpeg*, *image/pjpeg*, *image/x-png*, *image/png*, *image/svg+xml*.

---

delay option

The *delay* option lets you specify a delay (in ms) or a min-max range of delay (in milliseconds) after which the response will be sent.

If the given value is a number, it will be considered as a fix delay. You can also specify an object with a *min* and a *max* property which will respectively be the minimum and maximum delay time, in milliseconds.

Listing 46: YAML

```
http:
  request:
    path: '/foo'
  response:
    body: 'Hello, World!'
    delay:
      min: 20
      max: 500
```

Listing 47: JSON

```
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Hello, World!",
      "delay": {
        "min": 20,
        "max": 500
      }
    }
  }
}
```

headers option

The *headers* option lets you specify the headers sent along with the response. It is an object in which the key/value pairs correspond the name/value pairs of the headers.

Listing 48: YAML

```yaml
http:
  request:
    path: '/foo'
  response:
    body: 'Hello, World!'
    headers:
      Content-Type: 'text/plain'
```

Listing 49: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Hello, World!",
      "headers": {
        "Content-Type": "text/plain"
      }
    }
  }
}
```

status option

The *status* option lets you define the HTTP status code of the response.

Listing 50: YAML

```yaml
http:
  request:
    path: '/foo'
  response:
    body: 'Not found'
    status: 404
```

Listing 51: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "body": "Not found",
      "status": 404
    }
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

velocity option

Deprecated see Template.

template option

The *template* option either a boolean or an object. If a boolean, it tells mockservr if the value specified in *response.body* or *response.bodyFile* is an template file and should be parsed by the default template engine.

If an object, it may define an *enabled* value which tells mockservr if it should parse the response body as a template file. It may also define a *context* value which is an object. The values in this object will then be passed to the template engine and thus be available in the template file. It may also define an *engine* value which is a string which defined the engine to use for parse template file. available engines:

1. twig (default)
2. velocity

Velocity templates let you access some of the request's parameters (such as query params and form data) and forge a tailored response.

Listing 52: YAML

```yaml
http:
  request:
    path: '/foo'
  response:
    bodyFile: './responses/foo/response.json'
    template:
      enabled: true
      engine: velocity
```

Listing 53: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": {
      "bodyFile": "./responses/foo/response.json",
      "template": {
        "enabled": true,
        "engine": "velocity"
      }
    }
  }
}
```

Note: Mockservr take advantage of the *twig* template library. Check out their website for more information.

Note: Mockservr take advantage of the ‘VelocityJS‘VelocityJS_ javascript library, which does not implement all Velocity features. Check out their Github page for more information.

Note: More information about Apache Velocity template files can be found on Apache Velocity Documentation.

Note:

From within the Apache Velocity template, the following objects are available:

- a *math* object which is a Javascript *Math* object
- a *req* object which contains all data from the incoming HTTP request
- an *endpoint* which is the object representing the matched endpoint from the mock definition file
- a *context* which is the optional *velocity.context* object defined above

The *endpoint* object contains the parameters that were matched against the incoming HTTP request. It also contains information about how the parameter was matched (e.g: boolean or regex's capturing group).

weight option

In case you define multiple responses for a single Request, the *weight* option lets you put weights on responses so that the random selection of a response will be biased by this weight. Weight are plain numbers.

Listing 54: YAML

```yaml
http:
  request:
    path: '/foo'
  response:
    -
      body: 'Hello World!'
      weight: 10
    -
      body: "Bye World!"
      weight: 1
```

Listing 55: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo"
    },
    "response": [
      {
        "body": "Hello World!",
        "weight": 10
      },
      {
        "body": "Bye World!",
```

(continues on next page)

(continued from previous page)

```
          weight: 1
        }
      ]
    }
  }
```

In the previous example, the "Hello World!" response will be sent by Mockservr 10 out of 11 times and the "By World!" response 1 out of 11 times.

1.3 Validators

For an incoming HTTP to match a defined Request, it must be positively matched against all the options defined for the endpoint in the mock file.

To perform this, you may use a Validator ; it is an object with two properties:

- *type* which defines the type of Validator to use
- *value* which is the expected value used by the Validator.

Mockservr comes with a **Validator inference** feature, which means, if you do not explicitly define which Validator to use, Mockservr will guess it for you.

Inference transform rules :

- *string* will be transform in *equal* Validator
- *number* will be transform in *equal* Validator
- *boolean* will be transform in *equal* Validator
- *array* will be transform in *anyOf* Validator
- *null* will be transform in *object* Validator
- *object* that is not a validator will be transform in *object* Validator

1.3.1 *equal* Validator

The *equal* Validator performs an exact match between the expected value and the given one. This Validator is automatically inferred when the value is a string, a number or a boolean value.

Example

For example, you can use the *equal* Validator to validate the method. All the following definitions are equals:

Using the *equal* Validator

Listing 56: YAML

```
http:
  request:
    path: '/foo'
    method:
```

(continues on next page)

(continued from previous page)

```
        type: 'equal'
        value: 'GET'
    response:
      body: 'Hello World!'
```

Listing 57: JSON

```
{
  "http": {
    "request": {
      "path": "/foo",
      "method": {
        "type": "equal",
        "value": "GET"
      }
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

Using the Validator inference

Listing 58: YAML

```
http:
  request:
    path: "/foo"
    method: "GET"
  response:
    body: 'Hello World!'
```

Listing 59: JSON

```
{
  "http": {
    "request": {
      "path": "/foo",
      "method": "GET",
    },
    "response": {
      "body": "Hello World!"
    }
  }
}
```

1.3.2 *range* Validator

The *range* Validator may be used to define a range in which the given value should lie. The *value* is an object composed of two entries:

- *min*: The lower bound of the range (inclusive)

- *max*: The upper bound of the range (inclusive)

Both ranges must be numbers (either integer or floats). An example of the *range* Validator can be is presented in *query option*.

1.3.3 *regex* Validator

The *regex* Validator may be used to match a given value against a regular expression. As such, the *value* entry is the given regular expression. References about Javascript Regular Expressions can be found on Mozilla.

An example of the *regex* Validator can be is presented in *method option*.

1.3.4 *anyOf* Validator

The *anyOf* Validator may be used to match one of several given values. Under the hood, Mockservr is performing Validator inference ; it allows to use equals values (string, number, ...) in the array. However, it is possible to use Validators inside the array, giving you the possibility to use regular expression, for example.

Listing 60: YAML

```yaml
http:
  request:
    path: "/foo"
    method:
      type: 'anyOf'
      value:
        - 'GET'
        - 'POST'
        -
          type: 'regex'
          value: '/^P.*$/'
  response:
    body: 'Hello World!'
```

Listing 61: JSON

```json
{
  "http": {
    "request": {
      "path": "/foo",
      "method": {
        "type": "anyOf",
        "value": [
          "GET",
          "POST",
          {
            "type": "regex",
            "value": "/^P.*$/"
          }
        ]
      }
    },
    "response": {
      "body": "Hello World!"
    }
```

(continues on next page)

(continued from previous page)

```
      }
    }

  The endpoint is then available through different HTTP requests:
```

```
curl -XGET 'http://localhost:8080/foo'
curl -XPOST 'http://localhost:8080/foo'
curl -XPUT 'http://localhost:8080/foo'
```

1.3.5 *object* Validator

The *object* Validator in itself does not perform any validation. Instead, the value is an object, in which one or more validators are defined for each object attribute. Validation of *object* Validator will perform recursively.

1.3.6 *typeOf* Validator

The *typeOf* Validator validates that the given value corresponds to the expected type.

Listing 62: YAML

```yaml
    http:
      request:
        path: '/foo'
        method:
          type: 'typeof'
          value: 'string'
      response:
        body: 'Hello World!'
```

Listing 63: JSON

```
    {
      "http": {
        "request": {
          "path": "/foo"
          "method": {
            "type": "typeof",
            "value": "string"
          }
        },
        "response": {
          "body": "Hello World!"
        }
      }
    }

  The example above will match any incoming request, as method is always a string.
```

As Mockservr is using Javascript, running the *typeOf* validator against *null* won't be working as expected.
Use *object* Validator with *null* value instead.

1.4 API

Mockservr exposes an HTTP API which allow to get information about current endpoints, and to update them if needed. This API is available through HTTP queries on *http://localhost:4580* (the API is exposed through the port 4580 of the Mockservr's container).

1.4.1 API Endpoints

All endpoints are JSON endpoints (*Content-Type: application/json*) and must be prefixed with */api*.

/api endpoint

GET method

The response is an object with a single attribute *httpEndpoints*, it contains the number of endpoints currently served by Mockservr.

/api/http-endpoints endpoint

GET Method

The response is a collection of all HTTP endpoints cuurrently served by Mockservr. The response includes the internal ID of the endpoint and the source (mock file or API).

POST method

It expects a JSON body as defined in *HTTP Mocking*, defining an endpoint with a Request and a Response.

The response contains the newly created endpoint with its ID and source. If any error occurred, the response is an HTTP 400 response with a json object that contains all encountered errors.

/api/http-endpoints/:id endpoint

GET Method

The response is an object defining the endpoint corresponding to the given *:id*.

DELETE method

Deletes the endpoint from Mockservr. The response is an HTTP 204 response. If any error occurred, the response is an HTTP 400 response with a json object that contains all encountered errors.

Note: *DELETE* method does not delete the mock file, if the target endpoint is defined in a mock file.
