Mirage Documentation

Release 0.8.16 beta

OpenCredo

March 07, 2018

Contents

CHAPTER 1

Introduction

The target of the Mirage software is to enable automated testing by mastering system dependencies.

You can find out more about Mirage (inner data structures like scenarios, sessions and stubs) and how to use it in our _wiki: https://github.com/SpectoLabs/mirage/wiki/Introduction .

These docs store a bit more technical information like API reference and some examples how to integrate Mirage into your testing environment.

CHAPTER 2

REST API v2

Mirage REST API v2 returns JSON. The response always returns the version, and payload. Responses are similar to v1 API responses. The payload is either contained under 'data' if the response is successful or 'error' for errors. Errors contain a descriptive message under 'message' and the http error code under 'code'. Successful responses depend on the call made and are described below.

Encoding "application/json"

Create scenario object

Creates scenario object. Client must specify scenario name in the request body.

- **URL**: /api/v2/scenarios
- **Method**: PUT
- **Response codes**:
- **201** - scenario created
- **422** - scenario with that name already exists
- **400** - something is missing (e.g. name)
- **Example request body**:

```
{
  "scenario": "scenario_name"
}
```

Upload scenario with session & stubs

You can upload existing scenarios directly into Mirage via this API call. Currently it only accepts .zip format and only .yaml configuration file.

- **URL**: /api/v2/scenarios/upload
- **Method**: POST
- **Encoding**: "multipart/form-data"
- **Response codes**:
- **200** - Scenario uploaded successfully.

- **415** - Content type not supported (format not accepted).
- **400** - Configuration file found, however failed to read it.
- **422** - Scenario already exists.

Example scenario_x.zip content structure:

```
scenario_x.yaml
stub_0.json
stub_1.json
stub_2.json
stub_3.json
stub_4.json
```

As you can see - everything is expected to be in the same directory, no inner directory structure is expected.

Defining .yaml configuration:

```
recording:
  scenario: scenario_x
  session: session_x
```

Note that by default Mirage exported yaml file looks like this:

```
recording:
  scenario: scenario_x
  session: session_x
  stubs:
  - file: stub_0.json
  - file: stub_1.json
  - file: stub_2.json
  - file: stub_3.json
  - file: stub_4.json
```

Although, during upload it ignores “stubs” key and treats every .json file in the archived zip as a separate stub.

When Mirage finds yaml configuration - it gets scenario name and creates it. After that - reads every file from the archive that ends with ”.json”. After all the stubs were added to the database - starts a session in playback mode. Scenario is then ready for testing.

Example stub .json structure:

```
{
   "priority": 9903,
   "args": {
      "priority": "9903"
   },
   "request": {
      "bodyPatterns": {
         "contains": [
            "<tag> matcher here </tag>"
         ]
      },
      "method": "POST"
   },
   "response": {
      "body": "<response> Response here </response>\n",
      "headers":
      "status": 200
   }
}
```

Example upload (Python)

Below is an example pseudocode for uploading files to Mirage:

```
def upload(file_name, mirage_uri):
    """
    file_name - path to zip archive.
    mirage_uri - full Mirage URL (http://miragehostname:8001)
    """
    # reading file
    with open(file_name, 'r') as stream:
        # preparing data
        files = [('files', (file_name, stream, 'application/zip'))]
        data = {}

        resp = requests.post(mirage_uri + "/api/v2/scenarios/upload",
                             files=files, data=data)
        if resp.status_code == 200:
            print("Upload success!")
    return
```

You can upload multiple archives at once. Every archive should be self contained - yaml configuration and stubs archived together.

Get scenario list

Returns a list of scenarios with their name and scenarioRef attributes. This attribute can the be used to get scenario details (sessions, stubs, size, etc...). Your application can look for these keys and use them to directly access resource instead of generating URL on client side.

- **URL**: /api/v2/scenarios
- **Method**: GET
- **Response codes**:
- **200** - scenario list returned
- **Example output**:

```
    {
      "data": [
        {"scenarioRef": "/api/v2/scenarios/objects/localhost:scenario_1",
        "name": "localhost:scenario_1"},
        {"scenarioRef": "/api/v2/scenarios/objects/localhost:scenario_10",
         "name": "localhost:scenario_10"},
         ...,
         ...,
         ]
    }
```

Returns a list of scenarios with details.

- **URL**: /api/v2/scenarios/detail
- **Method**: GET
- **Response codes**:
- **200** - scenario list with details returned

- **Example output**:


```
{
  "data": [
  {
    "space_used_kb": 0,
    "name": "localhost:scenario_1",
    "sessions": [],
    "scenarioRef": "/api/v2/scenarios/objects/localhost:scenario_1",
    "recorded": "-",
    "stub_count": 0},
  {
    "space_used_kb": 544,
    "name": "localhost:scenario_10",
    "sessions": [
        {
          "status": "playback",
          "loaded": "2015-07-20",
          "name": "playback_10",
          "last_used": "2015-07-20 13:09:05"}
        ],
     "scenarioRef": "/api/v2/scenarios/objects/localhost:scenario_10",
     "recorded": "2015-07-20",
     "stub_count": 20},
     ...,
     ...,
   ]
}
```


Get scenario details

Returns specified scenario details. Scenario name can be provided with a host, for example stubo_c1.yourcorporateaddress.com:scenario_name_x

- **URL**: /api/v2/scenarios/objects/
- **Method**: GET
- **Response codes**:
- **200** - specified scenario details
- **404** - specified scenario not found
- **Example output**:


```
{
  "space_used_kb": 697,
  "name": "localhost:scenario_13",
  "sessions": [
    {"status": "playback",
    "loaded": "2015-07-20",
    "name": "playback_13",
     "last_used": "2015-07-20 13:09:05"}
   ],
   "stubs": 26,
   "recorded": "2015-07-20",
   "scenarioRef": "/api/v2/scenarios/objects/localhost:scenario_13"
 }
```

Delete scenario

Deletes scenario object and removed it's stubs from cache.

- **URL**: /api/v2/scenarios/objects/
- **Method**: DELETE
- **Response codes**:
- **200** - scenario deleted
- **412** - precondition failed - specified scenario does not exist

Begin session and set mode

Begins session for specified scenario. Client has to specify session name and mode in request body. Session mode can be either 'record' and 'playback'.

- **URL**: /api/v2/scenarios/objects//action
- **Method**: POST
- **Response codes**:
- **200** - begins session
- **400** - something went wrong (e.g. session already exists)
- **Example request body**:

```
{
  "begin": null,
  "session": "session_name",
  "mode": "record"
}
```

- **Example output**:

```
{
  "version": "0.6.6",
  "error":
     {"message": "Scenario (localhost:scenario_10) has existing stubs, delete them before record
     "code": 400}
}
```

End session

Ends specified session. Client has to specify session name in request body.

- **URL**: /api/v2/scenarios/objects//action
- **Method**: POST
- **Response codes**:
- **200** - session ended.
- **Example request body**:

```
    {
      "end": null,
      "session": "playback_28"
    }
```

- **Example output**:

```
    {
      "version": "0.6.6",
      "data": {
         "message": "Session ended"}
     }
```

End all sessions for specific scenario

Ends all sessions for specified scenario

- **URL**: /api/v2/scenarios/objects//action
- **Method**: POST
- **Response codes**:
- **200** - scenario list with details returned
- **Example request body**:

```
{
  "end": "sessions"
}
```

- **Example output**:

```
{
  "version": "0.6.6",
  "data": {
      "playback_20": {"message": "Session ended"}
      }
}
```

Add stub

Add stub to scenario

- **URL**: /api/v2/scenarios/objects/(?P[^\/]+)/stubs
- **Method**: PUT
- **Response codes**:
- **201** - inserted
- **200** - updated or ignored
- **Example request body**:

```
    {
        "request": {
            "method": "POST",
            "bodyPatterns": [
                { "contains": ["<status>IS_OK2</status>"] }
            ]
            },
        "response": {
            "status": 200,
            "body": "<response>YES</response>"
        }
    }
```

- **Example output**: If updated (status code 200)

```
    {
        version: "0.7"
        data: {
        message: "updated with stateful response"
        }
     }
```

or inserted (status code 201).

```
{
    version: "0.6.6"
    data: {
    message: "inserted scenario_stub: 55d5e7ebfc4562fb398dc697"
}
```

Here this ID - 55d5e7ebfc4562fb398dc697 is an object _id field from database. Proxy or an integrator could actually go directly to database with this key and retrieve response.

Getting response with stub

- **URL**: /api/v2/scenarios/objects/(?P[^\/]+)/stubs
- **Method**: POST
- **Response codes**:
- __*__ - any HTTTP response that user defined during stub insertion
- **Example request header**: session: your_session_name
- **Example request body**:

```
    matcher here
```

Getting request response data (JSON)

This call is similar to "Getting response with stub", however this one is intended to give all the data to the client to build original response instead of returning original response. This enables storing encoded/encrypted responses. Users have to provide a scenario:session key in headers.

- **URL**: /api/v2/matcher

- **Method**: POST
- **Response codes**:
- **200** - Response found
- **404** - Response not found
- **Example request header**: session: scenario_name:session_name
- **Example request body**:

```
    matcher here
```

- **Example response body**: ‘‘‘javascript { “body”: “body bytes or plain text here”, “headers”: { “key”: “value”, “key1”: “value1” } “statusCode”: 200 }

‘‘‘

Get all stubs for specific scenario

- **URL**: /api/v2/scenarios/objects/(?P[^\/]+)/stubs
- **Method**: GET
- **Response codes**:
- **200** - stub list returned

```
{
    "version": "0.7",
    "data": [
        {
            "stub": {
                "priority": -1,
                "request": {
                    "bodyPatterns": {
                        "contains": [
                            "<status>IS_OK2</status>"
                        ]
                    },
                    "method": "POST"
                },
                "args": {},
                "recorded": "2015-10-08",
                "response": {
                    "status": 123,
                    "body": "<response>YES</response>"
                }
            },
            "matchers_hash": "a92fa6cf96f218598d3723f2827a6815",
            "space_used": 219,
            "recorded": "2015-10-08",
            "scenario": "localhost:scenario_name_1"
        }
    ]
}
```

Delete all scenario stubs

- **URL**: /api/v2/scenarios/objects/(?P[^\/]+)/stubs
- **Method**: DELETE
- **Response codes**:
- **200** - stubs deleted
- **409** - precondition failed - there are active sessions either in playback or record mode

Get delay policy list

Gets all defined delay policies

- **URL**: /api/v2/delay-policy/detail
- **Method**: GET
- **Response codes**:
- **200** - list with delay policies returned
- **Example output**:

```
{
  "version": "0.6.6",
  "data": [
    {
      "delay_type": "fixed",
       "delayPolicyRef": "/api/v2/delay-policy/objects/slow",
      "name": "slow",
      "milliseconds": "1000"
    },
    {
      "delay_type": "fixed_2",
       "delayPolicyRef": "/api/v2/delay-policy/objects/slow",
      "name": "slower",
      "milliseconds": "5000"
    }
    ]
  }
```

Get specific delay policy details

- **URL**: /api/v2/delay-policy/objects/
- **Method**: GET
- **Response codes**:
- **200** - delay policy returned
- **404** - delay policy not found
- **Example output**:

```
{
  "version": "0.6.6",
  "data": [
    {
      "delay_type": "fixed",
      "delayPolicyRef": "/api/v2/delay-policy/objects/slow",
      "name": "slow",
      "milliseconds": "1000"
    }
  ]
}
```

Add delay policy

Creates a delay policy. Returns 201 status code if successful or 409 if request body options are wrong (type fixed provided with mean and stddev options)

- **URL**: /api/v2/delay-policy
- **Method**: PUT
- **Response codes**:
- **201** - scenario list with details returned
- **400** - bad request
- **409** - wrong combination of options was used
- **Example request body**:

```
    {
      "name": "delay_name",
      "delay_type": "fixed",
      "milliseconds": 50
    }
```

or:

```
{
  "name": "delay_name",
  "delay_type": "normalvariate",
  "mean": "mean_value",
  "stddev": "val"
}
```

or:

```
{
  "name": "weighted_delay_name",
  "delay_type": "weighted",
 "delays": "fixed,30000,5:normalvariate,5000,1000,15:normalvariate,1000,500,70"}
```

- **Example output**:

```
{
  "status_code": 201,
  "version": "0.6.6",
  "data":
```

```
  {
    "status": "new",
    "message": "Put Delay Policy Finished",
    "delay_type": "fixed",
    "delayPolicyRef": "/api/v2/delay-policy/objects/my_delay",
    "name": "my_delay"
  }
}
```


Delete delay policy

- **URL**: /api/v2/delay-policy/objects/
- **Method**: DELETE
- **Response codes**:
- **200** - delay policy deleted
- **Example output**:


```
{
  "version": "0.6.6",
  "data":
  {
    "message": "Deleted 1 delay policies from [u'my_delay']"
  }
}
```


Get records from tracker

Gets records from tracker. Since this collection becomes quite big over time - pagination is available. Your client application can define how many records it wants to skip and current item limit. Mirage also provides "href" - links to every record for additional information.

Available parameters: + limit - maximum records to return + skip - how many records should be skipped + q - query. Query can be simple string to search based on API call path, scenario. You can also supply values to search for specific status codes and response times. For example I would want to find a scenario named "my_test_suite_x" and to see only HTTP calls that produced status codes between 200 and 201, also, I would want to see only those requests that took longer than 200 ms to complete, my query would look like: "my_test_suite_x sc:>=200 sc:<=201 rt:>200"

- **URL**: /api/v2/tracker/records
- **Method**: GET
- **Response codes**:
- **200** - list with tracker entries returned
- **Example output**:


```
{
    "paging": {
        "last": "/api/v2/tracker/records?skip=23172&limit=2",
        "next": "/api/v2/tracker/records?skip=2&limit=2",
        "currentLimit": 2,
        "totalItems": 23174,
```

```
        "previous": null,
        "first": "/api/v2/tracker/records?skip=0&limit=2"
    },
    "data": [
        {
            "function": "/api/v2/scenarios/objects/localhost:scenario_100/stubs",
            "request_params": {},
            "start_time": "2015-10-27 11:18:24",
            "return_code": 200,
            "href": "/api/v2/tracker/records/objects/562f5d8137dd1220d73a0cbf",
            "duration_ms": 902,
            "stubo_response": "",
            "id": "562f5d8137dd1220d73a0cbf"
        },
        {
            "function": "/api/v2/scenarios/objects/localhost:scenario_14/action",
            "request_params": {},
            "scenario": "localhost:scenario_14",
            "start_time": "2015-10-27 10:15:01",
            "return_code": 200,
            "href": "/api/v2/tracker/records/objects/562f4ea537dd1220d73a0ca1",
            "duration_ms": 24,
            "stubo_response": "",
            "id": "562f4ea537dd1220d73a0ca1"
        }
    ]
}
```


Get tracker record details

Gets detail information about specified tracker object ID.

- **URL**: /api/v2/tracker/records/objects/
- **Method**: GET
- **Response codes**:
- **200** - tracker record found, returned.
- **404** - tracker record not found
- **Example output**:


```
{
   "data":{
      "function":"/api/v2/scenarios/objects/localhost:scenario_100/stubs",
      "request_params":{

      },
      "tracking_level":"normal",
      "start_time":"2015-10-27 11:18:24",
      "return_code":200,
      "server":"karoliss-macbook-pro.local",
      "request_size":0,
      "response_headers":{
         "Date":"Tue, 27 Oct 2015 11:18:24 GMT",
         "Content-Length":"5835405",
```

```
            "Content-Type":"application/json; charset=UTF-8",
            "Server":"TornadoServer/4.1"
         },
         "request_headers":{
            "Accept-Language":"en-US,en;q=0.8,lt;q=0.6",
            "Accept-Encoding":"gzip, deflate, sdch",
            "Connection":"keep-alive",
            "Accept":"*/*",
            "User-Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_0) AppleWebKit/537.36 (KHTML, li
            "Dnt":"1",
            "Host":"localhost:8001",
            "X-Requested-With":"XMLHttpRequest",
            "Referer":"http://localhost:8001/manage/scenarios/details?scenario=/api/v2/scenarios/objects
         },
         "host":"localhost",
         "request_method":"GET",
         "id":"562f5d8137dd1220d73a0cbf",
         "response_size":5835405,
         "duration_ms":902,
         "stubo_response":"",
         "port":"8001",
         "remote_ip":"::1"
      }
}
```

CHAPTER 3

API v1 (Legacy)

The Mirage API v1 returns JSON. The response always returns the version, and payload. The payload is either contained under 'data' if the response is successful or 'error' for errors. Errors contain a descriptive message under 'message' and the http error code under 'code'. Successful responses depend on the call made and are described below.

```
e.g. an error response

{
   "version": "1.2.3",
   "error": {
       "message": "Session already exists - localhost:first:first_1 in playback mode.",
       "code": 400
   }
}

e.g. a successful response

{
   "version": "1.2.3",
   "data": {
       "status": "playback",
       "message": "Playback mode initiated....",
       "session": "first_1",
       "scenario": "localhost:first"
   }
}
```

exec/cmds

```
 exec/cmds  (GET, POST)
    query args:
        cmdfile: URL or a file under /static on the mirage server
         + any user args will be made avaliable to the cmd file template
        session_id: session name to substitute within the cmdfile template, best to prefix this with
    response: shows the list of commands (url, return_code) executed, see the Tracker page for respon

Typically command files are used to load stubs into the Mirage db, but you can run any supported comm

stubo/api/exec/cmds?cmdfile=/static/cmds/demo/first.commands

{
```

```
"version": "1.2.3",
"data": {
   "executed_commands": [
      [
         "delete/stubs?scenario=first",
         200
      ],
      [
         "begin/session?scenario=first&session=first_1&mode=record",
         200
      ],
      [
         "put/stub?session=first_1,first.textMatcher,first.response",
         200
      ],
      [
         "end/session?session=first_1",
         200
      ],
      [
         "begin/session?scenario=first&session=first_1&mode=playback",
         200
      ],
      [
         "get/response?session=first_1,first.request",
         200
      ],
      [
         "end/session?session=first_1",
         200
      ]
   ],
   "number_of_requests": 7,
   "number_of_errors": 0
   }
}
```

```
stubo/api/exec/cmds?cmdfile=https://your-source-repo/my.commands

Command files can be included in an archive file along with the matcher and response files. You can
by running the archive file:

stubo/api/exec/cmds?cmdfile=https://your-source-repo/my.zip

Supported archive formats are zip, tar.gz & jar files.
```

get/version

```
get/version (GET, POST)

This call does not touch the db or cache so is useful as a quick 'ping' on the server

stubo/api/get/version
```

```
{"version": "1.2.3"}
```

get/status

```
get/status (GET, POST)
   query args:
     scenario=name
     session=name (session takes precedence)
     check_database=true|false (default true)
     local_cache=true|false (default true)

stubo/api/get/status?scenario=first

{
"version": "1.2.3",
"data": {
    "cache_server": {
        "status": "ok",
        "local": true
    },
    "info": {
        "cluster": "my-cluster",
        "graphite_host": "http://my-graphite.com/"
    },
    "database_server": {
        "status": "ok"
    },
    "sessions": [
        [
            "first_1",
            "dormant"
        ]
    ]
}

stubo/api/get/status?session=first_1

{
"version": "1.2.3",
"data": {
    "cache_server": {
        "status": "ok",
        "local": true
    },
    "info": {
        "cluster": "my-cluster",
        "graphite_host": "http://my-graphite.com/"
    },
    "session": {
        "status": "dormant",
        "system_date": "2014-10-02",
        "scenario": "localhost:first",
        "last_used": "2014-10-02 16:00:39",
        "scenario_id": "542d76a7ac5f73060fc9c2b4",
        "session": "first_1"
```

```
    },
    "database_server": {
        "status": "ok"
    }
}
```

begin/session

```
 begin/session (GET, POST)
    query args:
        scenario = scenario name
        session = session name
        mode = playback|record

stubo/api/begin/session?scenario=first&session=first_1&mode=playback

{
    "version": "5.9.9",
    "data": {
        "status": "playback",
        "message": "Playback mode initiated....",
        "session": "first_1",
        "scenario": "localhost:first"
    }
}

Note on duplicate scenarios and sessions:

* A scenario name prefixed with the mirage host name must be unique. One cannot record a new scenario
* Sessions are instances of scenario's stubs and must be unique within a host.
* Sessions can not be deleted if in playback or record mode
* Scenarios can not be deleted if any session based on it is in playback or record mode.
```

end/session

```
end/session (GET, POST)
   query args:
       session: session name

stubo/api/end/session?session=first_1

{
   "version": "1.2.3",
   "data": {
       "message": "Session ended"
   }
}

* Ending a session which does not exist is OK and will complete successfully
```

end/sessions

```
end/sessions (GET, POST)
   query args:
       scenario: scenario name

stubo/api/end/sessions?scenario=first

{
    "version": "6.1.3",
    "data": {
        "first_1": {
            "message": "Session ended"
        },
        "first_2": {
            "message": "Session ended"
        }
    }
}
```

put/scenarios

Scenario names can be changed by providing current scenario name and new name. This operation includes renaming all the stubs that belong to this scenario, as well as changing scenario name value in saved sessions. Sessions will be transfered to new scenario. During rename procedure - all sessions will be set to dormant mode. Returns status code 412 if no name or no query is provided. Returns status code 400 if scenario name has illegal characters. Scenario name check regex: r'[w-]*$' - letters, numbers, dashes, underscores

```
put/scenarios/(?P<scenario_name>[^\/]+) (GET)
   query args:
       new_name: new scenario name

stubo/api/put/scenarios/first?new_name=new_first_scenario_name

{
"Scenarios changed": 1,

"Remapped sessions": [
         {
             "name": "myscenario_session2"
         }
     ],
"New name": "localhost:new_first_scenario_name",
"Old name": "localhost:first",
"Stubs changed": 5,
"Pre stubs changed": 0
}
```

put/stub

```
 put/stub (POST)
    query args:
         session = session name
```

```
         ext_module = external module name without .py extenstion (optional)
         delay_policy =  delay policy name (optional)
         stateful = treat duplicate stubs as stateful otherwise ignore duplicates if stateful=false
         tracking_level: full or normal (optional, overrides host or global setting)
         + any user args will be made avaliable to the matcher & response templates and any user exit
 e.g.
 stubo/api/put/stub?session=my_session

 given request=<status>IS_OK</status> & response=<response>YES</response>
 JSON POST data
 {
     "request": {
         "method": "POST",
         "bodyPatterns": [
             { "contains": ["<status>IS_OK</status>"] }
         ]
         },
     "response": {
         "status": 200,
         "body": "<response>YES</response>"
     }
 }
 returns
 {
    "data": {
        "message": "put 54378c0dac5f7302b5cb8e56 stub"
    },
    "version": "1.2.3"
 }

 Treatment of duplicate stubs:

* If both the request and the response already exist for the scenario in record mode, then the stub w
* If the request exists, but with a different response, the second response will be recorded and the
* Duplicate stubs can exist in different scenarios
```

Notes:

see stub_reference for stub definitions. see daterolling for an example of using user arguments to perform date rolling

get/stublist

```
 get/stublist (GET, POST)
    query args:
        scenario: scenario name
        host: host uri to use (defaults to host used in request uri, optional)

 stubo/api/get/stublist?scenario=first

{
 "version": "1.2.3",
 "data": {
     "stubs": [
         {
             "recorded": "2014-10-10",
             "args": {
```

```
                "session": "first_1"
            },
            "request": {
                "bodyPatterns": [
                    {
                        "contains": [
                            "get my stub\n"
                        ]
                    }
                ],
                "method": "POST"
            },
            "response": {
                "status": 200,
                "body": "Hello {{1+1}} World\n"
            }
        }
    ],
    "scenario": "first"
}
```

put/delay_policy

```
put/delay_policy (GET, POST)
   query args:
       name: delay name
       delay_type: fixed, normalvariate or weighted
       milliseconds: used with fixed delay_type only
       mean: used with normalvariate delay_type only
       stddev: used with normalvariate delay_type only
       values: used with weighted delay_type only. values is a delimited string of delays.
       For each delay the last value represents the percentage this delay will occur.

stubo/api/put/delay_policy?name=slow&delay_type=fixed&milliseconds=1000

{
   "version": "1.2.3",
   "data": {
       "status": "new",
       "message": "Put Delay Policy Finished",
       "delay_type": "fixed",
       "name": "slow"
   }
}

i.e. to set a weighted percentage of delays with 5% fixed at 30s, 15% having a delay of 5s +/- 1s and
stubo/api/put/delay_policy?name=pcent_random_samples&delay_type=weighted&delays=fixed,30000,5:normalv

{
   "version": "1.2.3",
   "data": {
       "status": "new",
       "message": "Put Delay Policy Finished",
       "delay_type": "weighted",
       "name": "pcent_random_samples"
   }
```

```
}
```

get/delay_policy

```
get/delay_policy (GET, POST)
   query args:
       name: delay name (optional lists all if not provided)

stubo/api/get/delay_policy?name=slow
{
   "version": "1.2.3",
   "data": {
       "slow": {
           "delay_type": "fixed",
           "name": "slow",
           "milliseconds": "1000"
       }
   }
}
```

delete/delay_policy

```
delete/delay_policy (GET, POST)
   query args:
       name: delay name (optional deletes all if not provided)

stubo/api/delete/delay_policy?name=slow

{
   "version": "1.2.3",
   "data": {
       "message": "Deleted 1 delay policies from [u'slow']"
   }
}
```

get/response

```
get/response (POST)
   query args:
       session: session name
       tracking_level: full or normal (optional, overrides host or global setting)
   POST data: request payload
   HTTP headers:
     Mirage-Request-Session=123 Optional, can be used in place of session on the URL.
   returns stub response payload in HTTP body if ok
   on error returns mirage json error response

stubo/api/get/response?session=first_1
POST data: get my stub
returns: Hello 2 World
```

delete/stubs

Stubs should be mastered in a code repository such as SVN. Delete/stubs will remove stubs from the Mirage database. This should be run at the end of each test run.

```
 delete/stubs (GET, POST)
    query args:
        scenario: scenario name
        host: host uri to use (defaults to host used in request uri, optional)
        force: false or true (optional, defaults to false)

stubo/api/delete/stubs?scenario=first

{
    "version": "1.2.3",
    "data": {
        "scenarios": [
            "localhost:first"
        ],
        "message": "stubs deleted."
    }
}

* All sessions must be in a dormant state to delete the stubs unless force=true is used
* Deleting a scenario that does not exist is OK and will complete successfully
```

get/export

Export a recorded scenario. To support repeatable testing a recording should be exported with get/export and the resulting archive file saved to your source code repository (GIT etc). The exported archive contains all scenario stubs and a command script to reload them. The get/export call also supports exporting 'runnable' scenarios. A 'runnable' scenario will add a playback of a previous session to the command script. This can be useful to compare different test runs with each other.

```
get/export (GET, POST)
   query args:
       scenario: scenario name
       session_id: session id to use within the export (optional, defaults to epoch time)
       export_dir: export dir name (optional, defaults to scenario key)
       runnable: create a runnable scenario of a previous playback (optional)
       playback_session: playback session to use (required with runnable)
       session_id: session name to substitute within the cmdfile template (optional)
returns links to exported archive files (*.zip, *.tar.gz, *.jar)

stubo/api/get/export?scenario=first

{
   "version": "1.2.3",
   "data": {
       "scenario": "first",
       "export_dir_name": "/Users/rowan/dev/eclipse/workspace/stubo/static/exports/localhost_first",
       "links": [
           [
               "first_1412947560_0.response.0",
               "http://Rowan-MacBook-Pro-5.local:8001/static/exports/localhost_first/first_141294756
```

```
            ],
            [
                "first_1412947560_0_0.textMatcher",
                "http://Rowan-MacBook-Pro-5.local:8001/static/exports/localhost_first/first_141294756
            ],
            [
                "first.commands",
                "http://Rowan-MacBook-Pro-5.local:8001/static/exports/localhost_first/first.commands?v
            ],
            [
                "first.zip",
                "http://Rowan-MacBook-Pro-5.local:8001/static/exports/localhost_first/first.zip?v=34c1
            ],
            [
                "first.tar.gz",
                "http://Rowan-MacBook-Pro-5.local:8001/static/exports/localhost_first/first.tar.gz?v=8
            ],
            [
                "first.jar",
                "http://Rowan-MacBook-Pro-5.local:8001/static/exports/localhost_first/first.jar?v=34c1
            ]
        ]
    }
}

& runnable export

stubo/api/get/export?scenario=first&runnable=true&playback_session=first_1

{
    "version": "1.2.3",
    "data": {
        "runnable": {
            "last_used": {
                "start_time": "2015-03-24 16:57:03.248000+00:00",
                "remote_ip": "::1"
            },
            "playback_session": "first_1",
            "number_of_playback_requests": 1
        },
        "scenario": "first",
        "links": [
            [
                "first_1427285580_0.response.0",
                "http://vuze-on-pc2.home:8001/static/exports/localhost_first/first_1427285580_0.respo
            ],
            [
                "first_1427285580_0_0.textMatcher",
                "http://vuze-on-pc2.home:8001/static/exports/localhost_first/first_1427285580_0_0.tex
            ],
            [
                "first_1427285580_0.request",
                "http://vuze-on-pc2.home:8001/static/exports/localhost_first/first_1427285580_0.reque
            ],
            [
                "first.commands",
                "http://vuze-on-pc2.home:8001/static/exports/localhost_first/first.commands?v=98ad492
            ],
```

```
            [
                "first.zip",
                "http://vuze-on-pc2.home:8001/static/exports/localhost_first/first.zip?v=66a370b25ca2
            ],
            [
                "first.tar.gz",
                "http://vuze-on-pc2.home:8001/static/exports/localhost_first/first.tar.gz?v=da76a1ce2
            ],
            [
                "first.jar",
                "http://vuze-on-pc2.home:8001/static/exports/localhost_first/first.jar?v=66a370b25ca2
            ]
        ],
        "export_dir_path": "/Users/rowan/dev/eclipse/workspace/opencredo/stubo/latest/stubo-app/stubo
    }
}
```

get/stubcount

```
get/stubcount (GET, POST)
   query args:
       scenario: scenario name (optional)

Returns the number of stubs for a given scenario or all scenarios on host if
the scenario is not provided.

stubo/api/get/stubcount?scenario=first

{
   "version": "1.2.3",
   "data": {
       "count": 1,
       "scenario": "first"
   }
}
```

put/module

User exits can be applied to perform custom manipulation of Mirage matchers and responses. The user exits are python code defined with the UserExit API. The code is input into mirage with the following API call.

```
put/module (GET, POST)
   query args:
       name: full path to module can be a uri

stubo/api/put/module?name=/static/cmds/tests/ext/xslt/mangler.py

{
   "version": "1.2.3",
   "data": {
       "message": "added modules: ['localhost_mangler_v1']"
   }
}
```

Notes:

If the module code has not changed an error is returned indicating that the source has not changed otherwise a new version of the module is added to mirage dynamically.

get/modulelist

```
get/modulelist (GET, POST)
returns list of loaded modules

stubo/api/get/modulelist

{
   "version": "1.2.3",
   "data": {
       "info": {
           "mangler": {
               "loaded_sys_versions": [
                   "localhost_mangler_v1"
               ],
               "latest_code_version": 1
           }
       },
       "message": "list modules"
   }
}
```

delete/module

Delete named module.

```
 delete/module (GET, POST)
    query args:
        name: name of module without .py ext

{
    "version": "1.2.3",
    "data": {
        "deleted": [
            "localhost:mangler"
        ],
        "message": "delete modules: [u'mangler']"
    }
}
```

delete/modules

Delete all modules from this host URL.

```
delete/modules (GET, POST)

{
```

```
    "version": "6.1.3",
    "data": {
        "deleted": [
            "localhost:strip_ns",
            "localhost:ignore_dates",
        ],
        "message": "delete modules: ['strip_ns', 'ignore_dates']"
    }
}
```

Set Tracking Level

All API calls to Mirage will result in a tracking record being created. Default level tracking includes:

- start time
- duration
- any user configured delay
- mirage function
- return code and data
- session and scenario names
- response size
- server (Mirage server that handled the request)
- host (DNS of mirage used on the request)
- remote_ip (IP address of the client)

In addition, get/response calls can optionally force other items to be tracked including:

- matchers used
- matcher text before, during and after any mangling
- response text before, during and after any mangling

To enable/disable logging.

```
put/setting (GET, POST)
   query args:
       tracking_level=full or normal

stubo/api/put/setting?setting=tracking_level&value=full
{
   "version": "1.2.3",
   "data": {
       "new": "false",
       "host": "localhost",
       "all": false,
       "tracking_level": "full"
   }
}
```

Click on a get/response item in the Tracker page to see the full tracking data.

get/stats

Obtain the percent of get/response calls that are above a given latency value.

```
get/stats (GET, POST)
   query args:
       percent_above_value = threshold value in millisecs
       from=start time of metrics

e.g. to find the percent of Mirage responses that take more than 40ms (during the past 30min)

/stubo/api/get/stats?percent_above_value=40&from=-30mins

{
   "version": "5.6.2",
   "data": {
       "from": "-30mins",
       "target": "averageSeries(stats.timers.stubo.aws_cluster1.*.stuboapi.get_response.latency.mean_
       "metric": "latency",
       "to": "now",
       "percent_above_value": 40,
       "pcent": 0.0
   }
}

The key value being "pcent" which in this case is 0.0.
```

CHAPTER 4

Scenario Importing

Configuration files are used to load stub files and add behaviour to them (state, dates, etc.). These configuration files must obey .yaml syntax so libraries that were designed to load .yaml will always be able to modify them on the fly.

Stubs can also be loaded and tested using individual calls to the Mirage server using the Mirage HTTP API. Configuration files are a shorthand for the API, making it simpler to enable repeatable, automated interactions with Mirage by directly uploading scenario and setting it to playback mode for a particular test. All scenarios (with stubs that belong to them have to be archived into one zip file. Mirage will then find .yaml configuration file and based on it will create scenario object and session. It will treat every ".json" file found in the archive as a separate stub.

Configuration file (YAML)

An example

```
# Mirage YAML

# Describe your stubs here
recording:
  scenario: rest
  session:  rest_recording
```

Below is an example of stub json file.

stub1.json

```
{
       "request": {
           "method": "GET",
           "bodyPatterns": {
              "contains": [
                 "0"
              ]
           },
           "headers" : {
              "Content-Type" : "application/json",
              "X-Custom-Header" : "1234"
           }
       },
       "response": {
           "status": 200,
           "body": "{\"version\": \"1.2.3\", \"data\": {\"status\": \"all ok\"}}",
            "headers" : {
```

```
                "Content-Type" : "application/json",
                "X-Custom-Header" : "1234"
            }
        }
}
```

request1.json

```
{
        "request": {
            "method": "GET",
            "bodyPatterns": {
                "contains": [
                    "1"
                ]
            },
            "headers" : {
                "Content-Type" : "application/json",
                "X-Custom-Header" : "1234"
            }
        },
        "response": {
            "status": 200,
            "body": "{\"version\": \"1.2.3\", \"data\": {\"status\": \"all ok\"}}",
             "headers" : {
                "Content-Type" : "application/json",
                "X-Custom-Header" : "1234"
            }
        }
}
```

Note that these json payloads for the request and response are defined as strings.

CHAPTER 5

Mirage Command File (text format)

The line orientated text version of the command file is an old format. It does not support the matching required for REST. For example, you can't match on the request headers or method. It only supports one type of matcher which is the 'body contains' matcher. If the text specified for the 'body contains' matcher stub is found in the request a match is found. This is often however the only type of matcher required to successfully stub a system. This format is still supported for import via exec/cmds. If a text commands file is imported via a '.commands' text file on export via get/export a '.yaml' file will be exported.

This format is also not suitable for calling Mirage REST API since it can only translate calls into GET requests.

What is more, using this format to import scenarios will block Mirage HTTP server so it may affect other users who are running tests (server will not be accessible).

Text Command file

- Lines beginning with a '#' are treated as comments
- Commands follow the Mirage REST API, for example: begin/session?scenario=abc&session=12345&mode=record
- When the API requires something in the HTTP body, input can be sourced by listing files after the command (separated by commas). Files are assumed to be in the same directory as the command file. Alternatively a full URI can be provided to source the input (matcher, request or response).
- Blank lines are ignored.
- An example command is: put/stub?session=12345',feng_001.textMatcher, feng_002.textMatcher, feng_001.response. A put/stub command must have at least one matcher file and one response file. This example has two matchers.

What goes in the command files?

- A .textMatcher file could be some or all of the text from the request, e.g. AB1234</FlightInfoQuery>
- Multiple .textMatchers files may be used in a stub. All must evaluate to True against the request for a response to be returned.
- .request files contain the text of a request (e.g. text, xml, json). Note the using playback in commands is only used to test the stubs as these would be issued by the real system being tested in normal situations.
- .response files contain the text of a response (e.g. text, xml, json).
- Any naming convention can be used for matcher, request and response files. We have used suffixes of .textMatcher, .request & .response but you are free to use other extensions like matcher.xml, request.json, response.xml etc.

Command Scripting

Mirage supports the ability to load stubs from various and multiple sources. This means people can organise all the stubs that support a particular service or use case together, and any test that needs those stubs could pull them in from common stub libraries maximising stub reuse.

```
{% set foreign_url = 'http://www.google.co.uk' %}
{% set other_url = 'http://www.sun.com' %}
delete/stubs?scenario=smart
begin/session?scenario=smart&session=smart_1&mode=record
put/stub?session=smart_1,url={{other_url}},url={{foreign_url}}
put/stub?session=smart_1,text=random_rubble,text=response_text
end/session?session=smart_1
```

Command files are also programmable with Python code snippets. See the example below:

```
{% set session_name = 'smart_1' %}
delete/stubs?scenario=smart
begin/session?scenario=smart&session={{session_name}}&mode=record
put/stub?session={{session_name}},first.textMatcher,first.response
end/session?session={{session_name}}
begin/session?scenario=smart&session={{session_name}}&mode=playback
# make 3 requests
{% for i in range(0,3) %}
get/response?session={{session_name}},first.request
{% end %}
end/session?session={{session_name}}
```

Passing Arguments

Arguments can be passed into command files and used in them. The following example allows the session and scenario to be named externally to the command file.

```
exec/cmds?cmdFile=/static/cmds/tests/accept/smart_arg.commands&scen=bob&session=smart_1
```

The above call can be used as follows:

```
delete/stubs?scenario={{scen[0]}}
begin/session?scenario={{scen[0]}}&session={{ session[0] }}&mode=record
put/stub?session={{session[0]}},first.textMatcher,first.response
end/session?session={{session[0]}}
begin/session?scenario={{scen[0]}}&session={{session[0]}}&mode=playback
{% for i in range(0,3) %}
get/response?session={{session[0]}},first.request
{% end %}
end/session?session={{session[0]}}
```

CHAPTER 6

Delay policies, Templated responses, XML mangling, External modules

Add Delay to a Response

To emulate response times of systems being stubbed, one can set a delay on stubs. Delays are set on the stub response, not on the request to stubo. This is because one request into the system being tested can result in several back-end/stubo calls, each of which might need different delay settings.

Mirage provides reusable delay_policy objects which can be used across many stubs and across sessions. The delay_policy can be altered on the fly, during a load/performance test if desired. There are currently two delay_policy types to choose from. (1) fixed, and (2)random with mean and standard deviation.

A fixed delay policy can be created or modified with an API call such as:

```
put/delay_policy?name=a_fixed_delay_policy&delay_type=fixed&milliseconds=200
```

Note that the value for the 'name' can be set to any string and is used again in the put/stub. A variable delay can be set up as follows:

```
put/delay_policy?name=a_normalvariate_delay_policy&delay_type=normalvariate&mean=100&stddev=50
```

Allowed 'delay type' values are 'fixed' and 'normalvariate' The next step is to use the delay policy when loading stubs, eg.:

```
put/stub?session=abc&delay_policy=a_fixed_delay_policy
```

- Whenever the stub above is used, the named delay policy will be applied.
- Delay policies may be altered on the fly, during a test run if desired. To change a delay on the fly, simply repeat the put/delay_policy command with a different value. Typically this would be done directly by the test tool, rather than from a command file.
- Delay policies can be used across multiple sessions. This allows for easier alteration of policies where many sessions use the same policy. It also means that you should name policies carefully and only alter or share policies you are in control of.

Matching

Mirage currently supports various types of matchers.

Body contains matching

One or more matchers can be defined whereby all matchers must be contained in the request to return a specified response. A typical difficulty with system requests is that they often contain superfluous information such as session IDs or time stamps that get in the way of matching a request to the desired response. One way Mirage can solve this problem is by the use of 'matchers'. Matchers are parts of the request that matter to you, only what is needed to find the correct response file.

For example, a request may include:

```
<sessionid>123456</sessionid>
  <transactionid>0987654321-9999</transactionid>
      <departurestartdatetime>2009-10-24T00:00:00+00:00</departurestartdatetime>
            <flightnumber>0455</flightnumber>
```

Of these 4 lines only the last 2 matter, the first 2 lines will differ from one request to the next and be set by the system making the request. This request then will be different each time it is made. How can one easily match a request like this to a canned response file without writing code for each request/response pair ? The solution used by Mirage is to not store request/response file pairs, but rather store matchers and the response. In the case above the matchers would be:

```
<departurestartdatetime>2009-10-24T00:00:00+00:00</departurestartdatetime>
  <flightnumber>0455</flightnumber>
```

Mirage contains code which will take a request as input, search for the response with the best matchers. Every matcher must be contained in the request to be a match. When a match is found the corresponding response is served back.

All matchers must be present in the request to make a match. Mirage will return the response from the first match if there is more than one possible match. Note that whitespace and carriage returns are ignored when matching.

Alternatives to removing parts of the request you don't want to match on are

1. using a matcher template to substitute ids or dates etc from the request into the matcher OR
2. using a user exit to run custom code. A convenience class XMLManglerExit is provided for XML payloads to ignore XMl elements and/or attributes from the matching process.

Templated Matcher

This is an example of a template matcher that extracts values from the request into the matcher. The xmltree object is an lxml parsed root element passed into the template.

```
<request>
<transid>{{xmltree.xpath('/request/transid')[0].text}}</transid>
<dt>2009-10-24T00:00:00+00:00</dt>
<flightnumber>0455</flightnumber>
</request>
```

Templated Responses

A hand crafted Mirage response may contain variables which are substituted in. For example, a response may contain:

```
{% set flight_nbr='XX1234' %}
The flight number is: {{flight_nbr}}
```

The resulting response served wil be:

```
The flight number is: XX1234
```

Code may be used, for example adding random elements to a response.

```
{% import random %}
{% set rnd_nbr = random.randint(1,200) %}
Hello World {{ 1000 + rnd_nbr }}
```

The resulting response served will be:

```
Hello World - followed by a number between 1,000 and 1,200.
```

To put today's date in a response include the following:

```
{{today_str.format('%d%m%y')}}
```

Mirage responses are run through a Tornado template and any logic or commands allowed in these templates can be used. See http://www.tornadoweb.org/en/stable/template.html for details.

Request Data in Responses

Text Manipulation

It is possible to pull data from the stub request and put it in the response. Say you have 200 stubs that are identical apart from a userid. They could be condensed into, one stub which uses the userid from the request in the response. request:

```
<userid>abc123</userid>
```

matcher:

```
<userid>
```

response template (using Python):

```
Hello to {{request_text[8:14]}}
```

response:

```
Hello to abc123
```

XML Manipulation

Sometimes it is easier to use XPATH to pull data from a request and place it in a response. If you need the SessionId from this request:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wbs="http://xml.aaa.com/ws/2009/01/WBS_Session-2.0.xsd">
  <soapenv:Header>
     <wbs:Session>
        <wbs:SessionId>John1234</wbs:SessionId>
            ......
```

Pull the SessionId from the request and use it within the response template:

```
{% set namespaces = {'soapenv': 'http://schemas.xmlsoap.org/soap/envelope/',
   'wbs': 'http://xml.aaa.com/ws/2009/01/WBS_Session-2.0.xsd'}%}
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wbs="http://xml.aaa.com/ws/2009/01/WBS_Session-2.0.xsd">
  <soapenv:Header>
     <wbs:Session>
        <wbs:SessionId>{{xmltree.xpath('//soapenv:Header/wbs:Session/wbs:SessionId',
              namespaces=namespaces)[0].text}}</wbs:SessionId>
            ......
```

Note the xmltree variable is the parsed xml request (as an lxml Root object) made available to the template if the request is valid xml.

Another example - no namespaces in the request. Request:

```
<?xml version="1.0" encoding="UTF-8"?>
<CompensateCustomersCheck APCWCustom="-1" LocalCurrencyCode="GBP" NumberOfAdults="1">...
```

Response:

```
{% set currency_code = xmltree.xpath('/CompensateCustomersCheck')[0].attrib['LocalCurrencyCode'] %}
<response LocalCurrencyCode={{currency_code}}>pay me</response>
```

Stateful Stubs

State is important in simulating back-end systems. For example, if one was to check-in passenger Bob on a particular flight the back-end would allow this request to succeed only one time and subsequent attempts to check-in the same passenger on the same flight would be rejected. Mirage can emulate this functionality, or if desired for repeatable tests, Mirage can allow Bob to be checked in multiple times on the same flight.

Stateful stubbing is enabled automatically. If stubs are recorded or loaded in which the same request returns different responses Mirage will remember this and internally keep track of which response should come next.

If your test is to check-in Bob multiple times, simply do not load the stubs representing the check-in rejection and Mirage will allow multiple check-ins.

User Exits

Mirage provides hooks into the runtime to execute custom user code. Mirage can execute your custom code to transform stub matchers, requests and/or responses. This is a powerful mechanism when your stubs contain dynamic data or need to change over time. This can be useful to perform 'intelligent stubbing' for example to roll dates. The user exit API is contained in the stubo.ext.user_exit module.

API

XML Auto Mangling

If your data is XML a convenience class XMLUserExit is provided to enable auto mangling.

API

Hooks

Modules

Modules are added with the put/module command and referenced via the ext_module url arg in the put/stub call. For example:

```
put/module?name=/static/cmds/tests/ext/xslt/mangler.py
delete/stubs?scenario=mangler_xslt
begin/session?scenario=mangler_xslt&session=mangler_xslt1&mode=record
put/stub?session=mangler_xslt1&ext_module=mangler, first.request, first.response
end/session?session=mangler_xslt1
```

This example dynamically imports a test mangler from the Mirage server. Typically these external python modules that hook into the Mirage runtime would be sourced from source control system using a URL.

Note: user exit processing is only enabled if the 'ext_module' variable is set to a pre-loaded module during the put/stub call. Each Mirage host URL has its own copy of any loaded modules. Such modules are available to any test and, as such, should typically be loaded once per test suite.

Splitting

Useful to removing random elements from matchers: It can happen that the system under test will include seemingly random elements in the request. This will cause the next run of the test to not match. This problem can be solved by processing matchers when loading - during a record. A matcher can be split into two with the variable text left out. Future request must match both of the matchers to get the response. An example is:

```
<Cryptic_GetScreen_Query><Command>FXXNFCDXO/67678200990000770 6/1214/N1111*12345678:06/P1</Command></C
```

Which becomes two matchers.

```
<Cryptic_GetScreen_Query><Command>FXXNFCDXO/67678200990000770 6/1214/N1111*
```

and

```
:06/P1</Command></Cryptic_GetScreen_Query>
```

cutting out the random number '12345678' in the middle.

This matcher splitting can be done manually or automatically with a user exit. See /static/cmds/tests/ext/splitters/1/first.commands for and example. It uses the rules module at : /static/cmds/tests/ext/aaa.py

Multiple matchers excluding random request elements can be created manually and loaded as such:

```
put/stub?session=splitter_record, splitter_1.textMatcher, splitter_2.textMatcher, splitter.response
```

Caching Values

If emulating back-end behaviour means writing some code for a particularly tricky behaviour, Mirage exposes a simple key-value cache API to matchers and responses.

For example, if a response needs to increment a count each time it is used, you can define a module like this

```
import logging
from stubo.ext.user_exit import GetResponse, ExitResponse

log = logging.getLogger(__name__)

class IncResponse(GetResponse):
    '''
    Increments a value in the response using the provided cache.
    '''
    def __init__(self, request, context):
        GetResponse.__init__(self, request, context)

    def inc(self):
        cache = self.context['cache']
        val = cache.get('foo')
        log.debug('cache val = {0}'.format(val))
        if not val:
            val = 0
        val += 1
        cache.set('foo', val)
        return val

    def doResponse(self):
        stub = self.context['stub']
        stub.set_response_body('<response>{0}</response>'.format(self.inc()))
        return ExitResponse(self.request, stub)

def exits(request, context):
    if context['function'] == 'get/response':
        return IncResponse(request, context)
```

The lines above to note are:

```
val = cache.get('foo')

cache.set('foo', val)
```

You can set any key and value which is of use. As with other user extensions the example above can be loaded in a command with:

```
put/module?name=/static/cmds/tests/ext/cache/example.py
....
put/stub?session=cache_1&ext_module=example&tracking_level=full,1.request,1.response
```

CHAPTER 7

Change History

stubo Change History

Changelog

Added:

Changed:

Fixed:

Deprecated:

Removed:

0.8.16

Added: - Adding ability to parse XML response elements

0.8.15

Fixed: - Fixed yaml export for stubs with multiple responses

0.8.14

Added: - Adding ability to filter results by session id

0.8.13

Fixed: - Updated Mirage readthedocs.org documentation URL

0.8.12

Fixed: - Enabled SSL verification for HTTPS URLs.

0.8.11

Fixed: - Fixed tracker page not opening in IE

0.8.10

Changed: - Changed Stubo to respond with return header Content-Type if provided in the request

0.8.9

Changed: - Changed stubo-ba-ext to introduce ability to roll 8 digit dates

0.8.8

Added: - Added ability to delay the pulling stubo response into mememroy until after the set delay

0.8.7

Fixed: - Fixed a bug with unable to access tracker page in IE

0.8.6

Added:

- Fixed a bug with exporting scenarios not working for stubs with multiple responses
- Updated travis build URL for IAG Mirage repos.

0.8.5

Added:

- Added ability to explicitely format dates, for date rolling purposes
- Fixed a bug with multiple templated matches not evaluating propertly
- Fixed a bug with exporting not working correctly with external modules
- Fixed the documentation

0.8.4

Added:

- Now possible to delete scenarios with associated stubs, sessions etc. (https://github.com/SpectoLabs/BA-Mirage-Support/issues/5)
- Fixed display issues in tracker by ellipsizing. (https://github.com/SpectoLabs/BA-Mirage-Support/issues/3)
- Now able to restrict tracker logs to current host only. (https://github.com/SpectoLabs/BA-Mirage-Support/issues/11)

0.8.3 (2016-02-03)

Added: - Tutum configuration - Fixed bug when new external modules were added even though the source didn't change (https://github.com/SpectoLabs/mirage/pull/81)

0.8.2 (2015-11-05)

Added: - Displaying applied delays in event log (tracker) page if there was one. Users can now filter based on added delays

by adding "d:200" to display all 200 ms delays, operators like >, <, >=, <= are also available.

0.8.1 (2015-11-04)

Added: - Database authentication. Now you can specify MongoDB user (mongo.user) and password (mongo.password) in the .ini file. - Now looking for environment variables for database connection setup:

- export MONGO_URI=mongodb://<dbuser>:<dbpassword>@mongo_host:mongo_port/database
- export MONGO_DB=stubodb
- Cache (Redis) connection info can now be supplied through environment variables: + export REDIS_ADDRESS=new_hostname_or_ip + export REDIS_PORT=6379 + export REDIS_PASSWORD=very_secret More information can be found here: https://github.com/SpectoLabs/mirage/wiki/Configuration

Changed:

Fixed: - Track all hosts not showing information for all hostnames due to bug when loading initial cookie.

0.8.0 (2015-10-27)

Added: - Documentation for Tracker API and how to create useful queries. Also, created documentation for Tracker record details.

Changed: - Disabled certificate verification when importing scenarios (executing commands) - Changed all REST API (v2) URL paths: removed "/stubo/" from URL since it doesn't hold any information and there is no

point in providing application former name inside url.

- Updated documentation, tests to accommodate URL changes.

0.7.2 (2015-10-26)

Added: - Direct upload API (addresses https://github.com/SpectoLabs/mirage/issues/10). Now users can upload exported .zip files

directly into Mirage. There were also few modifications how Mirage parses .yaml configuration files such as now it only reads scenario and session names. All archived .json files will be treated as stubs and will be imported under the same scenario. This will make it easier to generate initial test data since you don't have to updated .yaml configuration file with each created stub file. Multiple PRs.

Changed: - Changed how Mirage exports scenarios to ".yaml" files. Now it doesn't add some non compatible tags like {{values}} to yaml file since

readers couldn't parse that information. Those lines were added to export files so it would be possible to change values during import but it is a lot easier to do that with standard "yaml" libraries.

- Delete scenario button is now disabled by default if scenario has active sessions in "playback" or "record" mode.
- Updates to documentation https://github.com/SpectoLabs/mirage/pull/62, https://github.com/SpectoLabs/mirage/pull/61.

Fixed: - Fixed bug when you could have deleted scenario which had active sessions. PR: https://github.com/SpectoLabs/mirage/pull/49

Removed: - Documentation cleanup in https://github.com/SpectoLabs/mirage/pull/62, https://github.com/SpectoLabs/mirage/pull/61. A lot

of content added to project wiki https://github.com/SpectoLabs/mirage/wiki. Since it is a lot easier to maintain wiki than sphinx docs - only essential, tightly coupled information like API reference and few other things were left in traditional docs.

0.7.1 (2015-10-20):

Added: - Added button to create scenario where users can also specify a session name and Mirage will start recording. - "End all sessions" button now instead of being disabled for dormant sessions - switches either to "begin playback"

or "start recording" based on current stub count.

- Added new API for getting information about found response (similar to get stub response) although instead of Mirage just using that information to create the original response - it now provides data about desired headers, body and status code. This enables proxy to store encoded data inside Mirage with required headers for decoding and then recreating original response without tampering with it. Original design was limited to plain text only.
- Users can now override session and scenario by supplying any hostname. So a user that is accessing Mirage through the hostname A - can also create scenario, start session and end it for host B. This was needed for running Mirage in container environment where communication between proxy and Mirage was established via DNS aliases (or links).
- Added all javascript libraries to source control to make development and management easier.

Changed: - By default user is now tracking all hosts (you can still choose to see only your host and that value will be saved in a cookie).

This was needed due to the fact that while running Mirage in a container - it' fairly difficult to know on which hostname Mirage is accessible through the container network or any other SDN.

- Date rolling by default now expects YYYY/MM/DD instead of YYYY/DD/MM.

Deprecated:

Removed: - Old design templates - Old web UI handlers, utilities (in service/api.py) - Old web UI javascript files and CSS styles

Fixed: - Fixed bug when due to websocket being blocked - tracker page failed to load new pages and filter. Now if it's available

in browser but terminated upon creation (i.e. firewall blocks it) - it will fall back to HTTP calls.

- When ensuring tracker indexes during startup - Mirage now reuses initial MongoDB connection, this solves some bugs when running in a container environment.
- Fixed problem when during startup Mirage also tried to import some functions from testing packages that required testing dependencies.

0.7 (2015-10-02)

Added: - REST API for Mirage management. This API currently covers most of the management commands

although it may slightly change (for example current "get response stub" request body is likely to be changed) This new API also has new functionality that was not available in the previous JSON API such as Tracker collection record filtering and pagination through REST. It also provides WebSocket supported filtering to speed up search in Tracker.

- Moved from Tornado template engine to React.js.
- New UI based on React.js framework. All web UI has been restructured, old /manage page was removed and its functionality split into "management" side menu where users can directly access scenario, module, delay policies or direct command execution pages. All of the data to this front-end is supplied by the new REST APIv2 and almost nothing is rendered on the server side (except for showing current Mirage version, however that may also change in future releases)
- Webpack configuration for compiling .jsx files to .js, this creates javascript bundles that can be easily imported in the separate pages and decreases loading time.
- Dockerfile for building Mirage Docker image and separate docker-compose script to launch dockerized Mirage environment that interconnects through container network.
- Syntax highlighting for

Changed: - Deployment and application startup has been changed. Now, instead of executing setup.py, copying configuration to

virtualenv and then from there launching Mirage by calling "stubo" command (that was far from standard approach at launching Python application) - it was changed to installing requirements from requirements.txt file with pip and then executing "python run.py" in the project root.

- Tracker collection is now being checked during each application startup. Mirage now checks whether it exists and can create a capped collection if it' not there. Then it ensures indexes.

Deprecated: - Current web UI is currently deprecated, although not removed. - Current handlers, handlers_mt, api functionality for serving and legacy UI is deprecated.

Fixed:

- Multiple open connections to MongoDB during tests
- Performance degradation due to indexing fixed
- Fault in multi-date rolling (BA issue)

0.6.6 (2015-08-10)

- Fixed bug when due to variable imports from testing packages startup failed
- Imports refactored, modules are now more independent

0.6.5 (2015-07-24)

- Correctly showing weighted delays in /manage page #77
- Moving python package requirements into separate requirements file, added production and development requirement files to avoid installing unnecessary testing packages in production. #78

0.6.4 (2015-07-16)

- Added indexes to database, optimized queries to database - /manage page is loading quickly now even when there are tens of thousands of stubs and tracker collection larger than > 100 000 records. #75
- Switched to mongo aggregation framework functionality when querying scenarios for their sizes and record dates. #68
- Added API call to rename scenario (should improve automated testing as it is a solution to several GitHub issues raised by BA). #71
- Optimised stub insertion #69
- Brought back export to .commands file, new YAML format files are exported in separate directory #64
- Fixed several bugs that could have been causing memory overflow in Stubo instances. #61 #62
- Parameter overrides when uploading stubs
- "worst latency" chart

0.6.3 (2015-06-30)

- YAML config (#12)
- Fix to support repeating elements within XMLMangler (#55)
- New weighted delay (#45)

0.6.2

- Rest support
- Parameterize session on get/export and exec/cmds calls (#39)

0.6.1 (2015-05-11)

- Return the http status code of exec/cmds requests (#28)
- Added tracker scenario filter (#23)
- Added stub priority
- Export runnable scenarios

0.6 (2015-03-25)

- Initial cut of open source version

CHAPTER 8

Indices and tables

- genindex
- modindex
- search