Molotov Documentation

Release 2.6

Tarek Ziadé

Oct 21, 2022

Contents

Simple Python 3.7+ tool to write load tests.

Based on asyncio, aiohttp.client.

CHAPTER 1

Quickstart

To create a load test, you need to create a Python module with some functions decorated with the **scenario** decorator – those functions needs to be coroutines. When executed, the function receives a **session** object inherited from **aiohttp.ClientSession**.

Here's a full example :

```
"""

This Molotov script has 2 scenario

"""
from molotov import scenario


_API = "http://localhost:8080"


@scenario(weight=40)
async def scenario_one(session):
    async with session.get(_API) as resp:
        res = await resp.json()
        assert res["result"] == "OK"
        assert resp.status == 200


@scenario(weight=60)
async def scenario_two(session):
    async with session.get(_API) as resp:
        assert resp.status == 200
```

When molotov runs, it creates some workers (coroutines) that will run the functions indefinitely until the test is over. A test is over when the time in seconds provided by **–duration** option is reached or when the maximum number of runs defined by **–max-runs** is reached.

By default, molotov will run for 86400 seconds (one day), but you can stop it any time with Ctrl-C or by sending a

TERM signal to the process.

Each worker randomly picks one scenario to execute, given their weights. Once it's finished, it picks the next one, and so on. In our example, **scenario_two** is picked ~60% of the time.

Note: Check out aiohttp's documentation to understand how to work with a session object.

Link: https://aiohttp.readthedocs.io/en/stable/client.html

To run the load script, you can provide its module name or its path. In the example below, the script is executed with 10 processes and 200 workers for 60 seconds:

```
$ molotov molotov/tests/example.py -p 10 -w 200 -d 60
```

When you run molotov, it will start a console application that will display the activity of each worker on the left panel. When a worker fails, the error will be displayed on the right panel

The footer will display a count of all successful and failed scenario runs, along with the current number of active workers (across all processes) and processes.

CHAPTER 2

Next steps

Check out the detailed documentation:

2.1 Installation

Make sure you are using Python 3.5 or superior with Pip installed, then:

```
$ pip install molotov
```

2.2 Command-line options

To run a test, use the **molotov** runner and point it to the scenario module or path:

Load test.

```
usage: molotov [-h] [--single-run] [-s SINGLE_MODE] [--config CONFIG]
               [--version] [--debug] [-v] [-w WORKERS] [--ramp-up RAMP_UP]
               [--sizing] [--sizing-tolerance SIZING_TOLERANCE]
               [--delay DELAY] [--console-update CONSOLE_UPDATE]
               [-p PROCESSES] [-d DURATION] [-r MAX_RUNS] [-q] [-x] [-f FAIL]
               [-c] [--statsd] [--statsd-address STATSD_ADDRESS] [--uvloop]
               [--use-extension USE_EXTENSION [USE_EXTENSION ...]]
               [--force-shutdown]
               [scenario]
```

2.2.1 Positional Arguments

scenario path or module name that contains scenarii

Default: “loadtest.py”

2.2.2 Named Arguments

--single-run	Run once every existing scenario Default: False
-s, --single-mode	Name of a single scenario to run once.
--config	Point to a JSON config file.
--version	Displays version and exits. Default: False
--debug	Run the event loop in debug mode. Default: False
-v, --verbose	Verbosity level. -v will display tracebacks. -vv requests and responses. Default: 0
-w, --workers	Number of workers Default: 1
--ramp-up	Ramp-up time in seconds Default: 0.0
--sizing	Autosizing Default: False
--sizing-tolerance	Sizing tolerance Default: 5.0
--delay	Delay between each worker run Default: 0.0
--console-update	Delay between each console update Default: 0.2
-p, --processes	Number of processes Default: 1
-d, --duration	Duration in seconds Default: 86400
-r, --max-runs	Maximum runs per worker
-q, --quiet	Quiet Default: False
-x, --exception	Stop on first failure. Default: False
-f, --fail	Number of failures required to fail
-c, --console	Use simple console for feedback Default: False

--statsd	Activates statsd
	Default: False
--statsd-address	Statsd Address
	Default: "udp://localhost:8125"
--uvloop	Use uvloop
	Default: False
--use-extension	Imports a module containing Molotov extensions
--force-shutdown	Cancel all pending workers on shutdown
	Default: False

2.3 API & Fixtures

To write tests, the only function you need to use is the `scenario()` decorator.

`molotov.`**`scenario`**(*weight=1*, *delay=0.0*, *name=None*)

Decorator to register a function as a Molotov test.

Options:

- **weight** used by Molotov when the scenarii are randomly picked. The functions with the highest values are more likely to be picked. Integer, defaults to 1. This value is ignored when the *scenario_picker* decorator is used.
- **delay** once the scenario is done, the worker will sleep *delay* seconds. Float, defaults to 0. The general –delay argument you can pass to Molotov will be summed with this delay.
- **name** name of the scenario. If not provided, will use the function __name___ attribute.

The decorated function receives an `aiohttp.ClientSession` instance.

If you don't want scenarii to be picked randomly given the weights, you can provide your own scenario picker function, by decorating it with the `scenario_picker()` decorator.

`molotov.`**`scenario_picker`**()

Called to chose a scenario.

Arguments received by the decorated function:

- **worker_id** the worker number
- **step_id** the loop counter

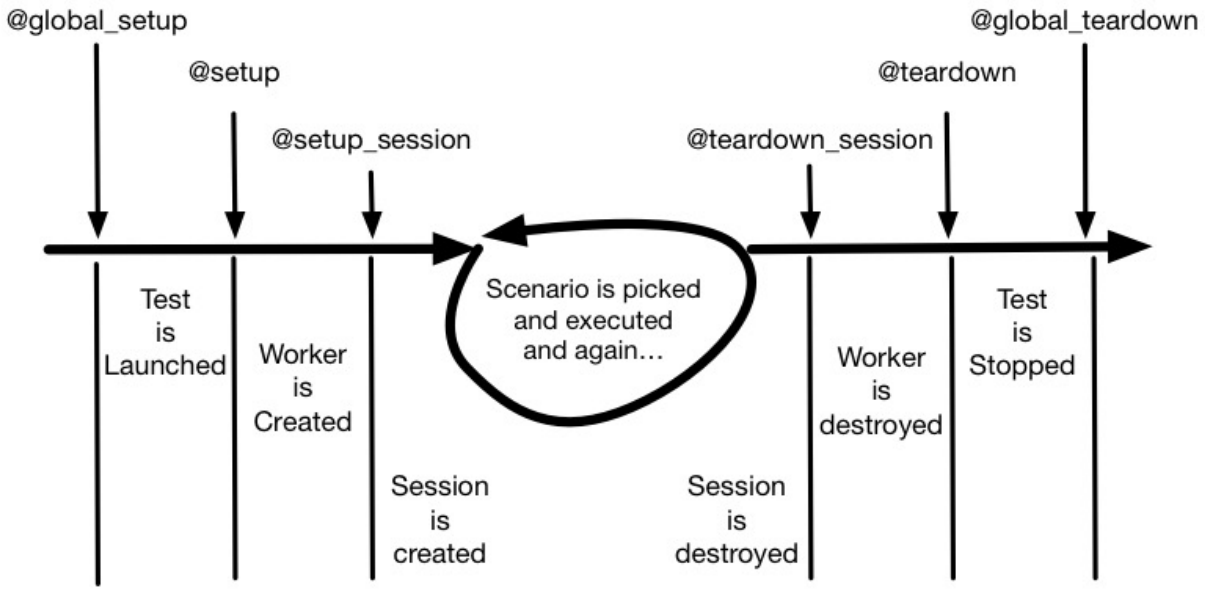
The decorated function should return the name of the scenario the worker should execute next.

When used, the weights are ignored.

The decorated function should not be a coroutine.

Molotov also provides optional decorators to deal with test fixtures.

The lifecycle of a test is shown in the diagram below, and test fixtures can be used to run functions at various stages.



molotov.**global_setup**()
Called once when the test starts.

The decorated function is called before processes and workers are created.

Arguments received by the decorated function:

- **args** arguments used to start Molotov.

This decorator is useful if you need to set up some fixtures that are shared by all workers.

The decorated function should not be a coroutine.

molotov.**setup**()
Called once per worker startup.

Arguments received by the decorated function:

- **worker_id** the worker number
- **args** arguments used to start Molotov.

The decorated function can send back a dict. This dict will be passed to the `aiohttp.ClientSession` class as keywords when it's created.

This is useful when you need to set up session-wide options like Authorization headers, or do whatever you need on startup.

The decorated function should be a coroutine.

molotov.**setup_session**()
Called once per worker startup.

Arguments received by the decorated function:

- **worker_id** the worker number
- **session** the `aiohttp.ClientSession` instance created

The function can attach extra attributes to the session and use **session.loop** if needed.

It's a good place to attache an object that interacts with the event loop, so you are sure to use the same one that the session's.

The decorated function should be a coroutine.

`molotov.`**`teardown_session`**`()`
Called once per worker when the session is closing.

Arguments received by the decorated function:

- **worker_id** the worker number
- **session** the `aiohttp.ClientSession` instance

The decorated function should be a coroutine.

`molotov.`**`teardown`**`()`
Called when a worker is done.

Arguments received by the decorated function:

- **worker_id** the worker number

The decorated function should not be a coroutine.

`molotov.`**`global_teardown`**`()`
Called when everything is done.

The decorated function should not be a coroutine.

Here's a full example, in order of calls:

```
"""

This Molotov script has:

- a global setup fixture that sets variables
- an init worker fixture that sets the session headers
- an init session that attachs an object to the current session
- 1 scenario
- 2 tear downs fixtures

"""
import molotov


class SomeObject(object):
    """Does something smart in real life with the async loop."""

    def __init__(self, loop):
        self.loop = loop

    def cleanup(self):
        pass


@molotov.global_setup()
def init_test(args):
    molotov.set_var("SomeHeader", "1")
    molotov.set_var("endpoint", "http://localhost:8080")
```

(continues on next page)

(continued from previous page)

```
@molotov.setup()
async def init_worker(worker_num, args):
    headers = {"AnotherHeader": "1", "SomeHeader": molotov.get_var("SomeHeader")}
    return {"headers": headers}


@molotov.setup_session()
async def init_session(worker_num, session):
    molotov.get_context(session).attach("ob", SomeObject(loop=session.loop))


@molotov.scenario(100)
async def scenario_one(session):
    endpoint = molotov.get_var("endpoint")
    async with session.get(endpoint) as resp:
        res = await resp.json()
        assert res["result"] == "OK"
        assert resp.status == 200


@molotov.teardown_session()
async def end_session(worker_num, session):
    molotov.get_context(session).ob.cleanup()


@molotov.teardown()
def end_worker(worker_num):
    print("This is the end for %d" % worker_num)


@molotov.global_teardown()
def end_test():
    print("This is the end of the test.")
```

2.4 Helpers

Molotov provides a few helpers to make it easier to write tests.

2.4.1 Global variables

If you need to use an object in various test fixtures or tests, you can use the `set_var()` and `get_var()` functions.

`molotov.`**`set_var`**(*name*, *value*)
Sets a global variable.

Options:

- name: name of the variable
- value: object to set

`molotov.`**`get_var`**(*name*, *factory=None*)
Gets a global variable given its name.

If factory is not None and the variable is not set, factory is a callable that will set the variable.

If not set, returns None.

2.4.2 Synchronous requests

If you need to perform synchronous requests in your setup:

molotov.**request**(*endpoint*, *verb='GET'*, *session_options=None*, ***options*)
Performs a synchronous request.

Uses a dedicated event loop and aiohttp.ClientSession object.

Options:

- endpoint: the endpoint to call
- verb: the HTTP verb to use (defaults: GET)
- session_options: a dict containing options to initialize the session (defaults: None)
- options: extra options for the request (defaults: None)

Returns a dict object with the following keys:

- content: the content of the response
- status: the status
- headers: a dict with all the response headers

molotov.**json_request**(*endpoint*, *verb='GET'*, *session_options=None*, ***options*)
Like *molotov.request()* but extracts json from the response.

```
from molotov import global_setup, json_request, set_var


@global_setup(args)
def _setup():
    set_var('token', json_request('http://example.com')['content'])
```

2.5 Events

You can register one or several functions to receive events emited during the load test. You just need to decorate the function with the *molotov.events()* fixture described below:

molotov.**events**()
Called everytime Molotov sends an event

Arguments received by the decorated function:

- **event** Name of the event
- extra argument(s) specific to the event

The decorated function should be a coroutine.

IMPORTANT This function will directly impact the load test performances

Current supported events and their keyword arguments:

- **sending_request**: session, request

- **response_received**: session, response, request
- **current_workers**: workers
- **scenario_start**: scenario, wid
- **scenario_success**: scenario, wid
- **scenario_failure**: scenario, exception, wid

The framework will gradually get more events triggered from every step in the load test cycle.

In the example below, all events are printed out:

```
"""

This Molotov script demonstrates how to hook events.

"""

import molotov


@molotov.events()
async def print_request(event, **info):
    if event == "sending_request":
        print("=>")


@molotov.events()
async def print_response(event, **info):
    if event == "response_received":
        print("<=")


@molotov.scenario(100)
async def scenario_one(session):
    async with session.get("http://localhost:8080") as resp:
        res = await resp.json()
        assert res["result"] == "OK"
        assert resp.status == 200
```

2.6 Extending Molotov

Molotov has a **–use-extension** option that can be used to load one or several arbitrary Python modules that contains some fixtures or event listeners.

Using extensions is useful when you want to implement a behavior that can be reused with arbitrary load tests.

In the example below `record_time()` is used to calculate the average response time of the load test:

```
"""

This Molotov script show how you can print
the average response time.

"""
import molotov
```

(continues on next page)

(continued from previous page)

```
import time


_T = {}


def _now():
    return time.time() * 1000


@molotov.events()
async def record_time(event, **info):
    req = info.get("request")
    if event == "sending_request":
        _T[req] = _now()
    elif event == "response_received":
        _T[req] = _now() - _T[req]


@molotov.global_teardown()
def display_average():
    average = sum(_T.values()) / len(_T)
    print("\nAverage response time %dms" % average)
```

When a Molotov test uses this extension, the function will collect execution times and print out the average response time of all requests made by Molotov:

```
$ molotov --use-extension molotov/tests/example6.py --max-runs 10 loadtest.py -c
Loading extension '../molotov/tests/example6.py'
Preparing 1 worker...
OK
[W:0] Starting
[W:0] Setting up session
[W:0] Running scenarios

Average response time 16ms
**** Molotov v2.6. Happy breaking! ****
SUCCESSES: 10 | FAILURES: 0
*** Bye ***
```

2.7 Run from GitHub

To run **molotov** directly from a GitHub repo, add a **molotov.json** at the top of that repo alongside your molotov tests.

molotov.json is a configuration file that contains a list of tests to run. Each test is defined by a name and the options that will be passed in the command line to **molotov**.

In the following example, three tests are defined: **test** and **big** and **scenario_two_once**:

```
{
    "molotov": {
        "env": {
            "SERVER_URL": "http://aserver.net"
        },
        "requirements": "requirements.txt",
```

(continues on next page)

(continued from previous page)

```
        "tests": {
            "big": {
                "console": true,
                "duration": 10,
                "exception": true,
                "processes": 10,
                "scenario": "molotov/tests/example.py",
                "workers": 100
            },
            "fail": {
                "exception": true,
                "max_runs": 1,
                "scenario": "molotov/tests/example3.py"
            },
            "scenario_two_once": {
                "console": true,
                "exception": true,
                "max_runs": 1,
                "scenario": "molotov/tests/example.py",
                "single_mode": "scenario_two"
            },
            "test": {
                "console": true,
                "duration": 1,
                "exception": true,
                "verbose": 1,
                "console_update": 0,
                "scenario": "molotov/tests/example.py"
            }
        }
    }
}
```

Once you have that file on the top of you repository, you can directly run it using **molotov**, with the **moloslave** command.

Example:

```
$ moloslave https://github.com/tarekziade/molotov test
```

This will simply run **molotov** with the options from the json file.

As demonstrated in example, there are also two global options you can use when running the tests:

- **requirements**: points to a Pip requirements file that will be installed prior to the test
- **env**: mapping containing environment variables that will be set prior to the test

2.8 Run in Docker

If your test can *Run from GitHub*, we provide a generic Docker image in the Docker Hub called **molotov**, that can be used to run your load test inside Docker.

The Docker image will use Moloslave against a provided repository. It's configured with two environment variables:

- **TEST_REPO** – the Git repository (has to be public)

- **TEST_NAME** – the name of the test to run

Example:

```
docker run -i --rm -e TEST_REPO=https://github.com/tarekziade/molotov -e TEST_
↪NAME=test tarekziade/molotov:latest
```

2.9 Step-by-step tutorial

Load testing a service with Molotov is done by creating a Python script that contains **scenarii**. A scenario is a somewhat realistic interaction with the service a client can have.

Before you can do anything, make sure you have Python 3.5+ and virtualenv.

Let's use **molostart** to get started:

```
$ molostart
**** Molotov Quickstart ****

Answer to a few questions to get started...
> Target directory [.]: /tmp/mytest
> Create Makefile [y]:
Generating Molotov test...
...copying 'Makefile' in '/tmp/mytest'
...copying 'loadtest.py' in '/tmp/mytest'
...copying 'molotov.json' in '/tmp/mytest'

All done. Happy Breaking!
Go in '/tmp/mytest'
Run 'make build' to get started...
```

molostart creates a default molotov layout for you. You can build the test with **make build** it will create a virtualenv inside the directory with Molotov installed.

```
$ cd /tmp/mytest
$ make build
$ source venv/bin/activate
(venv)
```

If that worked, you should now have a **molotov** command-line.

```
(venv) $ molotov --version
2.6
```

2.9.1 Running one scenario

Let's open loadtests.py, remove all the examples, and create our first real load test:

```
from molotov import scenario

@scenario(weight=100)
async def _test(session):
    async with session.get('https://example.com') as resp:
        assert resp.status == 200, resp.status
```

Molotov is used by marking some functions with the @scenario decorator. A scenario needs to be a coroutine and gets a **session** instance that can be used to query a server.

In our example we query https://example.com and make sure it returns a 200. Let's run it in console mode just once with *–single-run -c*:

```
(venv) $  molotov --single-run -c loadtest.py
**** Molotov v2.0. Happy breaking! ****
Preparing 1 workers...OK
SUCCESSES: 1 | FAILURES: 0  | WORKERS: 1
*** Bye ***
```

It worked!

Note: If you get a cryptic *certificate verify failed* error, make sure your Python installation has the root SSL certificates installed. This is usually done by installing *certifi*

Let's try for 30 seconds now with the console application (so without *-c*).

```
(venv) $  molotov -d 30 -x loadtest.py
```

You should see a nice application UI with live updates.

```
molotov/ex % venv/bin/molotov loadtest.py -d 30 -x
Molotov v2.6 ~ Happy Breaking 🥛🔨 ~ Ctrl+C to abort

[W:0] Ran 280 scenarios
[W:0] Ran 290 scenarios
[W:0] Ran 300 scenarios
[W:0] Ran 310 scenarios
[W:0] Ran 320 scenarios
[W:0] Ran 330 scenarios
[W:0] Ran 340 scenarios
[W:0] Ran 350 scenarios
[W:0] Ran 360 scenarios
[W:0] Ran 370 scenarios
[W:0] Ran 380 scenarios
[W:0] Ran 390 scenarios
[W:0] Ran 400 scenarios
[W:0] Ran 410 scenarios
[W:0] Ran 420 scenarios
[W:0] Ran 430 scenarios
[W:0] Ran 440 scenarios
[W:0] Ran 450 scenarios
[W:0] Ran 460 scenarios
[W:0] Ran 470 scenarios
[W:0] Ran 480 scenarios

SUCCESS: 475 FAILED: 0 WORKERS: 1 PROCESSES: 1 ELAPSED: 8.89 seconds
```

Notice that you can stop the test anytime with Ctrl+C.

The next step is to add more workers with -w. A worker is a coroutine that will run the scenario concurrently. Let's run the same test with 10 workers:

```
(venv) $ molotov -w 10 -d 30 -x loadtest.py
```

Molotov can also run several processes in parallel, each one running its own set of workers. Let's try with 4 processes and 10 workers. Virtually it means the level of concurrency will be 40:

```
(venv) $ molotov -w 10 -p 4 -d 30 -x loadtest.py
```

You can usually raise the number of workers to a few hundreds, and the number of processes to a few dozens. Depending how fast the server responds, Molotov can reach several thousands requests per second.

2.9.2 Adding more scenarii

You can add more scenarii and adapt their weights:

```
from molotov import scenario

@scenario(weight=20)
async def _test(session):
    async with session.get('https://example.com') as resp:
        assert resp.status == 200, resp.status

@scenario(weight=20)
async def _test2(session):
    # do something

@scenario(weight=60)
async def _test3(session):
    # do something different
```

The weights (20/20/60) define how often a scenario is executed by a worker. These weights does not have to be a sum of 100. Molotov will simply use this formula to determine how often a scenario is used:

```
scenario_weight / sum(scenario weights)
```

2.9.3 Adding test fixtures

Test fixtures are useful when you need to call a function once before the tests start, and when you want to configure the worker's session for all calls that will be made with it.

For instance, if you need an Authorization header that's shared across all workers and processes, you can use `global_setup()` to bake it and `setup()` to pass it to the session object that's created for each worker:

```
from molotov import setup, global_setup, scenario

_HEADERS = {}


@global_setup()
def init_test(args):
    _HEADERS['Authorization'] = 'Token xxxx'

@setup()
```

(continues on next page)

(continued from previous page)

```
async def init_worker(worker_id, args):
    return {'headers': _HEADERS}
```

Notice that the function decorated by `setup()` needs to be a coroutine.

2.9.4 Autosizing

Molotov comes with an autosizing feature. When the –sizing option is used, Molotov will slowly ramp-up the number of workers per process and will stop once there are too many failures per minute.

The default tolerance for failure is 5%, but this can be tweaked with the –sizing-tolerance option.

Molotov will use 500 workers that are getting ramped up in 5 minutes, but you can set your own values with –workers and –ramp-up if you want to autosize at a different pace.

```
(venv) $ molotov --sizing loadtest.py
```

2.10 Examples

These scripts are examples we use in the tests.

You can find them in https://github.com/tarekziade/molotov/tree/master/molotov/tests

```
"""

This Molotov script has:

- a global setup fixture that sets a global headers dict
- an init worker fixture that sets the session headers
- 3 scenario
- 2 tear downs fixtures

"""
import json
from molotov import (
    scenario,
    setup,
    global_setup,
    global_teardown,
    teardown,
    get_context,
)


_API = "http://localhost:8080"
_HEADERS = {}


# notice that the global setup, global teardown and teardown
# are not a coroutine.
@global_setup()
def init_test(args):
    _HEADERS["SomeHeader"] = "1"
```

(continues on next page)

(continued from previous page)

```
@global_teardown()
def end_test():
    print("This is the end")


@setup()
async def init_worker(worker_num, args):
    headers = {"AnotherHeader": "1"}
    headers.update(_HEADERS)
    return {"headers": headers}


@teardown()
def end_worker(worker_num):
    print("This is the end for %d" % worker_num)


# @scenario(weight=40)
async def scenario_one(session):
    async with session.get(_API) as resp:
        if get_context(session).statsd:
            get_context(session).statsd.incr("BLEH")
        res = await resp.json()
        assert res["result"] == "OK"
        assert resp.status == 200


@scenario(weight=30)
async def scenario_two(session):
    async with session.get(_API) as resp:
        assert resp.status == 200


# @scenario(weight=30)
async def scenario_three(session):
    somedata = json.dumps({"OK": 1})
    async with session.post(_API, data=somedata) as resp:
        assert resp.status == 200
```

```
"""

This Molotov script has 2 scenario

"""
from molotov import scenario


_API = "http://localhost:8080"


@scenario(weight=40)
async def scenario_one(session):
    async with session.get(_API) as resp:
        res = await resp.json()
        assert res["result"] == "OK"
```

(continues on next page)

(continued from previous page)

```
        assert resp.status == 200


@scenario(weight=60)
async def scenario_two(session):
    async with session.get(_API) as resp:
        assert resp.status == 200
```

```
"""

This Molotov script has:

- a global setup fixture that sets variables
- an init worker fixture that sets the session headers
- an init session that attachs an object to the current session
- 1 scenario
- 2 tear downs fixtures

"""
import molotov


class SomeObject(object):
    """Does something smart in real life with the async loop."""

    def __init__(self, loop):
        self.loop = loop

    def cleanup(self):
        pass


@molotov.global_setup()
def init_test(args):
    molotov.set_var("SomeHeader", "1")
    molotov.set_var("endpoint", "http://localhost:8080")


@molotov.setup()
async def init_worker(worker_num, args):
    headers = {"AnotherHeader": "1", "SomeHeader": molotov.get_var("SomeHeader")}
    return {"headers": headers}


@molotov.setup_session()
async def init_session(worker_num, session):
    molotov.get_context(session).attach("ob", SomeObject(loop=session.loop))


@molotov.scenario(100)
async def scenario_one(session):
    endpoint = molotov.get_var("endpoint")
    async with session.get(endpoint) as resp:
        res = await resp.json()
        assert res["result"] == "OK"
        assert resp.status == 200
```

(continues on next page)

(continued from previous page)

```
@molotov.teardown_session()
async def end_session(worker_num, session):
    molotov.get_context(session).ob.cleanup()


@molotov.teardown()
def end_worker(worker_num):
    print("This is the end for %d" % worker_num)


@molotov.global_teardown()
def end_test():
    print("This is the end of the test.")
```

```
"""

This Molotov script demonstrates how to hook events.

"""

import molotov


@molotov.events()
async def print_request(event, **info):
    if event == "sending_request":
        print("=>")


@molotov.events()
async def print_response(event, **info):
    if event == "response_received":
        print("<=")


@molotov.scenario(100)
async def scenario_one(session):
    async with session.get("http://localhost:8080") as resp:
        res = await resp.json()
        assert res["result"] == "OK"
        assert resp.status == 200
```

```
"""

This Molotov script show how you can print
the average response time.

"""
import molotov
import time


_T = {}
```

(continues on next page)

(continued from previous page)

```
def _now():
    return time.time() * 1000


@molotov.events()
async def record_time(event, **info):
    req = info.get("request")
    if event == "sending_request":
        _T[req] = _now()
    elif event == "response_received":
        _T[req] = _now() - _T[req]


@molotov.global_teardown()
def display_average():
    average = sum(_T.values()) / len(_T)
    print("\nAverage response time %dms" % average)
```

```
"""

This Molotov script uses events to display concurrency info

"""
import molotov
import time


concurs = []  # [(timestamp, worker count)]


def _now():
    return time.time() * 1000


@molotov.events()
async def record_time(event, **info):
    if event == "current_workers":
        concurs.append((_now(), info["workers"]))


@molotov.global_teardown()
def display_average():
    print("\nconcurrencies: %s", concurs)
    delta = max(ts for ts, _ in concurs) - min(ts for ts, _ in concurs)
    average = sum(value for _, value in concurs) * 1000 / delta
    print("\nAverage concurrency: %.2f VU/s" % average)
```

```
"""

This Molotov script uses events to generate a success/failure output

"""
import json
import molotov
import time
```

(continues on next page)

(continued from previous page)

```
_T = {}


def _now():
    return time.time() * 1000


@molotov.events()
async def record_time(event, **info):
    if event == "scenario_start":
        scenario = info["scenario"]
        index = (info["wid"], scenario["name"])
        _T[index] = _now()
    if event == "scenario_success":
        scenario = info["scenario"]
        index = (info["wid"], scenario["name"])
        start_time = _T.pop(index, None)
        duration = int(_now() - start_time)
        if start_time is not None:
            print(
                json.dumps(
                    {
                        "ts": time.time(),
                        "type": "scenario_success",
                        "name": scenario["name"],
                        "duration": duration,
                    }
                )
            )
    elif event == "scenario_failure":
        scenario = info["scenario"]
        exception = info["exception"]
        index = (info["wid"], scenario["name"])
        start_time = _T.pop(index, None)
        duration = int(_now() - start_time)
        if start_time is not None:
            print(
                json.dumps(
                    {
                        "ts": time.time(),
                        "type": "scenario_failure",
                        "name": scenario["name"],
                        "exception": exception.__class__.__name__,
                        "errorMessage": str(exception),
                        "duration": duration,
                    }
                )
            )
```

Index