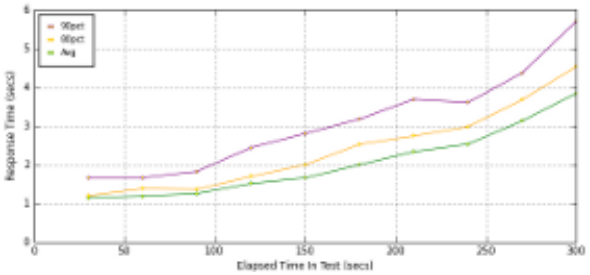
Multi-Mechanize - Performance Test Framework

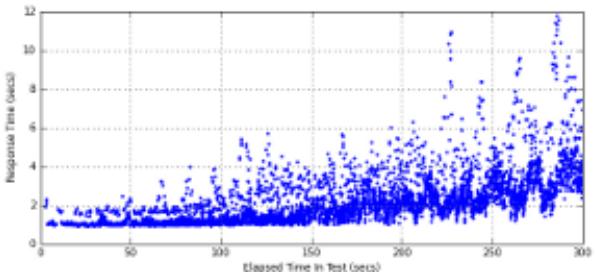
Release 1.2.0.1

Nov 09, 2017

Contents

Multi-Mechanize





Web multimechanize.com

PyPI multi-mechanize package

Dev GitHub

License GNU LGPLv3

Author Corey Goldberg - copyright © 2010-2013

CHAPTER 1

Performance & Load Tests in Python

Multi-Mechanize is an open source framework for performance and load testing. It runs concurrent Python scripts to generate load (synthetic transactions) against a remote site or service.

Multi-Mechanize is most commonly used for web performance and scalability testing, but can be used to generate workload against any remote API accessible from Python.

Test output reports are saved as HTML or JMeter-compatible XML.

CHAPTER 2

Site Menu

2.1 Detailed Install and Setup

The following instructions are for Debian/Ubuntu Linux. For other platforms, the setup is generally the same, with the exeption of installing system dependencies.

2.1.1 required

Multi-Mechanize requires Python **2.6** or **2.7**

2.1.2 system-wide install

- install dependencies on Debian/Ubuntu:

```
$ sudo apt-get install python-pip python-matplotlib
```

- install multi-mechanize from PyPI using Pip:

```
$ sudo pip install -U multi-mechanize
```

2.1.3 virtualenv + pip install (with matplotlib system package)

- install dependencies on Debian/Ubuntu:

```
$ sudo apt-get install python-virtualenv python-matplotlib
```

- install multi-mechanize from PyPI in a virtualenv:

```
$ virtualenv --system-site-packages ENV
$ cd ENV
$ source bin/activate
(ENV)$ pip install multi-mechanize
```

2.1.4 virtualenv + pip install (with no-site-packages)

- install dependencies on Debian/Ubuntu:

```
$ sudo apt-get install build-essential libfreetype6-dev libpng-dev
$ sudo apt-get install python-dev python-virtualenv
```

- install multi-mechanize and matplotlib from PyPI in a virtualenv:

```
$ virtualenv ENV
$ cd ENV
$ source bin/activate
(ENV)$ pip install multi-mechanize
(ENV)$ pip install matplotlib
```

2.1.5 pip install latest dev branch from git repo

```
pip install -e git+http://github.com/cgoldberg/multi-mechanize.git#egg=multimechanize
```

2.2 Configuration

2.2.1 Config File (config.cfg)

Each project contains a `config.cfg` file where test settings are defined.

The config file contains a `[global]` section and `[user_group-*]` sections.

2.2.2 Minimal Configuration

Here is a sample `config.cfg` file showing minimal options, defining 1 group of virtual users:

```
[global]
run_time = 100
rampup = 100
results_ts_interval = 10

[user_group-1]
threads = 10
script = vu_script.py
```

2.2.3 Full Configuration

Here is a sample `config.cfg` file showing all possible options, defining 2 groups of virtual users:

```
[global]
run_time = 300
rampup = 300
results_ts_interval = 30
progress_bar = on
console_logging = off
xml_report = off
results_database = sqlite:///my_project/results.db
post_run_script = python my_project/foo.py

[user_group-1]
threads = 30
script = vu_script1.py

[user_group-2]
threads = 30
script = vu_script2.py
```

2.2.4 Global Options

The following settings/options are available in the `[global]` config section:

- `run_time`: duration of test (seconds) [required]
- `rampup`: duration of user rampup (seconds) [required]
- `results_ts_interval`: time series interval for results analysis (seconds) [required]
- `progress_bar`: turn on/off console progress bar during test run [optional, default = on]
- `console_logging`: turn on/off logging to stdout [optional, default = off]
- `xml_report`: turn on/off xml/jtl report [optional, default = off]
- `results_database`: database connection string [optional]
- `post_run_script`: hook to call a script at test completion [optional]

2.2.5 User Groups

The following settings/options are available in each `[user_group-*]` config section:

- `threads`: number of threads/virtual users
- `script`: virtual user test script to run

2.3 Scripting Guide

2.3.1 Virtual User Script Writing

Scripts are written in Python. You have full access to the Python standard library and any additional modules you have installed.

2.3.2 The Basics

Each script must implement a `Transaction()` class. This class must implement a `run()` method.

So a basic test script consists of:

```
class Transaction(object):
    def run(self):
        # do something here
        return
```

During a test run, your `Transaction()` class is instantiated once, and then its `run()` method is called repeatedly in a loop:

```
class Transaction(object):

    def __init__(self):
        # do per-user user setup here
        # this gets called once on user creation
        return

    def run(self):
        # do user actions here
        # this gets called repeatedly
        return
```

2.3.3 Basic Examples

A full user script that generates HTTP GETs, using `mechanize`:

```
import mechanize

class Transaction(object):
    def run(self):
        br = mechanize.Browser()
        br.set_handle_robots(False)
        resp = br.open('http://www.example.com/')
        resp.read()
```

This script adds response assertions:

```
import mechanize

class Transaction(object):
    def run(self):
        br = mechanize.Browser()
        br.set_handle_robots(False)

        resp = br.open('http://www.example.com/')
        resp.read()

        assert (resp.code == 200), 'Bad Response: HTTP %s' % resp.code
        assert ('Example Web Page' in resp.get_data())
```

This script uses a custom timer. Custom timers are used to wrap blocks of code for timing purposes:

```
import mechanize
import time

class Transaction(object):
    def run(self):
        br = mechanize.Browser()
        br.set_handle_robots(False)

        start_timer = time.time()
        resp = br.open('http://www.example.com/')
        resp.read()
        latency = time.time() - start_timer

        self.custom_timers['Example_Homepage'] = latency
```

2.3.4 Advanced Examples

Wikipedia search with form fill/submit, timers, assertions, custom headers, think-times:

```
import mechanize
import time


class Transaction(object):

    def __init__(self):
        pass

    def run(self):
        # create a Browser instance
        br = mechanize.Browser()
        # don't bother with robots.txt
        br.set_handle_robots(False)
        # add a custom header so wikipedia allows our requests
        br.addheaders = [('User-agent', 'Mozilla/5.0 Compatible')]

        # start the timer
        start_timer = time.time()
        # submit the request
        resp = br.open('http://www.wikipedia.org/')
        resp.read()
        # stop the timer
        latency = time.time() - start_timer

        # store the custom timer
        self.custom_timers['Load_Front_Page'] = latency

        # verify responses are valid
        assert (resp.code == 200), 'Bad Response: HTTP %s' % resp.code
        assert ('Wikipedia, the free encyclopedia' in resp.get_data())

        # think-time
        time.sleep(2)

        # select first (zero-based) form on page
        br.select_form(nr=0)
```

```
        # set form field
        br.form['search'] = 'foo'

        # start the timer
        start_timer = time.time()
        # submit the form
        resp = br.submit()
        resp.read()
        # stop the timer
        latency = time.time() - start_timer

        # store the custom timer
        self.custom_timers['Search'] = latency

        # verify responses are valid
        assert (resp.code == 200), 'Bad Response: HTTP %s' % resp.code
        assert ('foobar' in resp.get_data()), 'Text Assertion Failed'

        # think-time
        time.sleep(2)
```

this example generates HTTP GETs, using `urllib2`:

```
import urllib2
import time

class Transaction(object):
    def run(self):
        start_timer = time.time()
        resp = urllib2.urlopen('http://www.example.com/')
        content = resp.read()
        latency = time.time() - start_timer

        self.custom_timers['Example_Homepage'] = latency

        assert (resp.code == 200), 'Bad Response: HTTP %s' % resp.code
        assert ('Example Web Page' in content), 'Text Assertion Failed'
```

this example generates HTTP POSTs containing a SOAP request in its body, using `urllib2`. The request message (SOAP envelope) is read from a file:

```
import urllib2
import time

class Transaction(object):
    def __init__(self):
        self.custom_timers = {}
        with open('soap.xml') as f:
            self.soap_body = f.read()

    def run(self):
        req = urllib2.Request(url='http://www.foo.com/service', data=self.soap_body)
        req.add_header('Content-Type', 'application/soap+xml')
        req.add_header('SOAPAction', 'http://www.foo.com/action')

        start_timer = time.time()
        resp = urllib2.urlopen(req)
        content = resp.read()
```

```
        latency = time.time() - start_timer

        self.custom_timers['Example_SOAP_Msg'] = latency

        assert (resp.code == 200), 'Bad Response: HTTP %s' % resp.code
        assert ('Example SOAP Response' in content), 'Text Assertion Failed'
```

this example generates HTTP POSTs, using `httplib`:

```
import httplib
import urllib
import time


class Transaction(object):
    def __init__(self):
        self.custom_timers = {}

    def run(self):
        post_body=urllib.urlencode({
            'USERNAME': 'corey',
            'PASSWORD': 'secret',})
        headers = {'Content-type': 'application/x-www-form-urlencoded'}

        start_timer = time.time()
        conn = httplib.HTTPConnection('www.example.com')
        conn.request('POST', '/login.cgi', post_body, headers)
        resp = conn.getresponse()
        content = resp.read()
        latency = time.time() - start_timer

        self.custom_timers['LOGIN'] = latency
        assert (resp.status == 200), 'Bad Response: HTTP %s' % resp.status
        assert ('Example Web Page' in content), 'Text Assertion Failed'
```

this example generates HTTP GETs, using httplib, with detailed timing:

```
import httplib
import time

class Transaction(object):
    def run(self):
        conn = httplib.HTTPConnection('www.example.com')
        start = time.time()
        conn.request('GET', '/')
        request_time = time.time()
        resp = conn.getresponse()
        response_time = time.time()
        conn.close()
        transfer_time = time.time()

        self.custom_timers['request sent'] = request_time - start
        self.custom_timers['response received'] = response_time - start
        self.custom_timers['content transferred'] = transfer_time - start

        assert (resp.status == 200), 'Bad Response: HTTP %s' % resp.status
```

```
if __name__ == '__main__':
    trans = Transaction()
    trans.run()

    for timer in ('request sent', 'response received', 'content transferred'):
        print '%s: %.5f secs' % (timer, trans.custom_timers[timer])
```

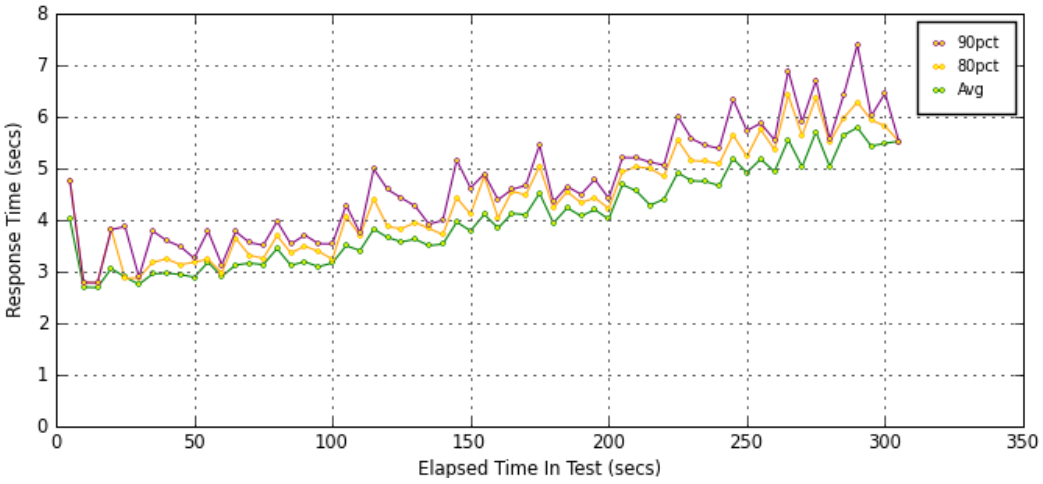
2.4 Sample Reports and Graphs

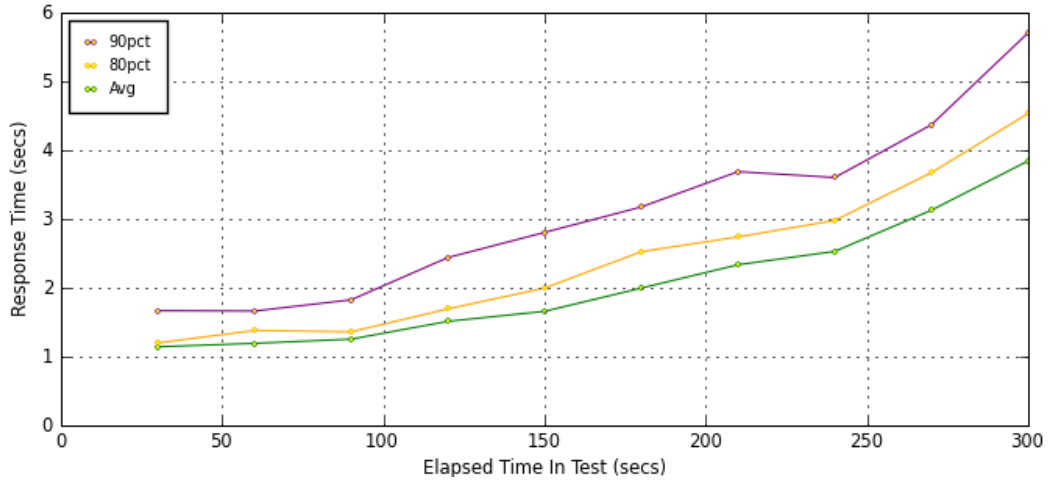
2.4.1 Sample Results Reports

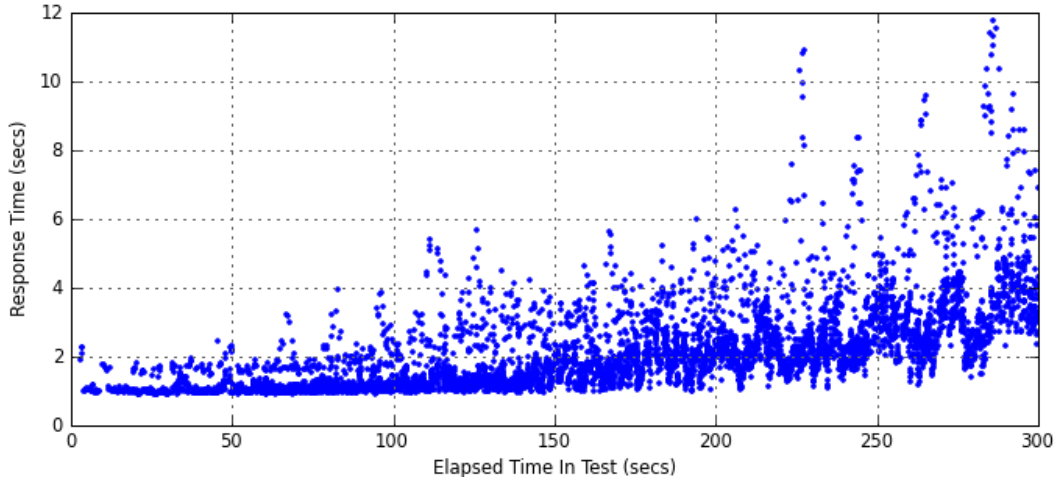
- results.html 1
- results.html 2
- results.html 3

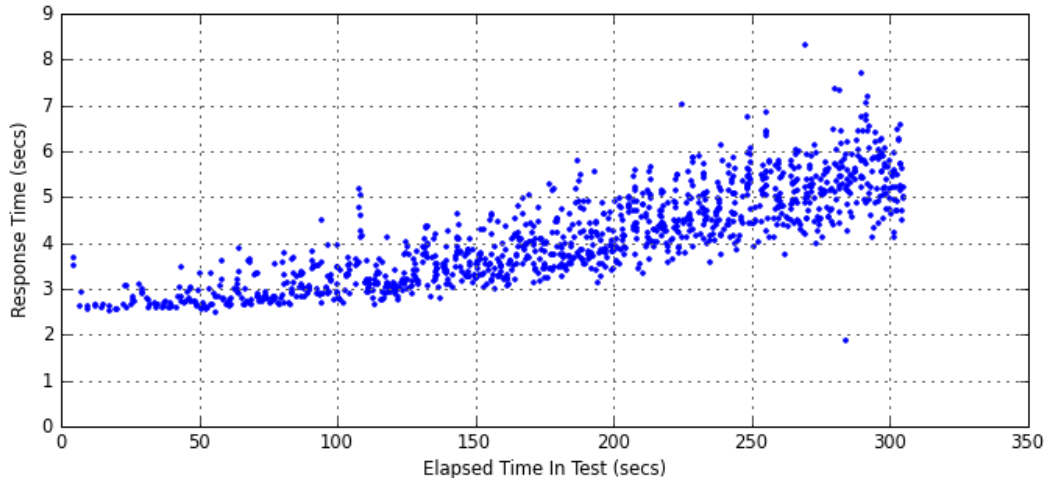
2.4.2 Sample Graphs

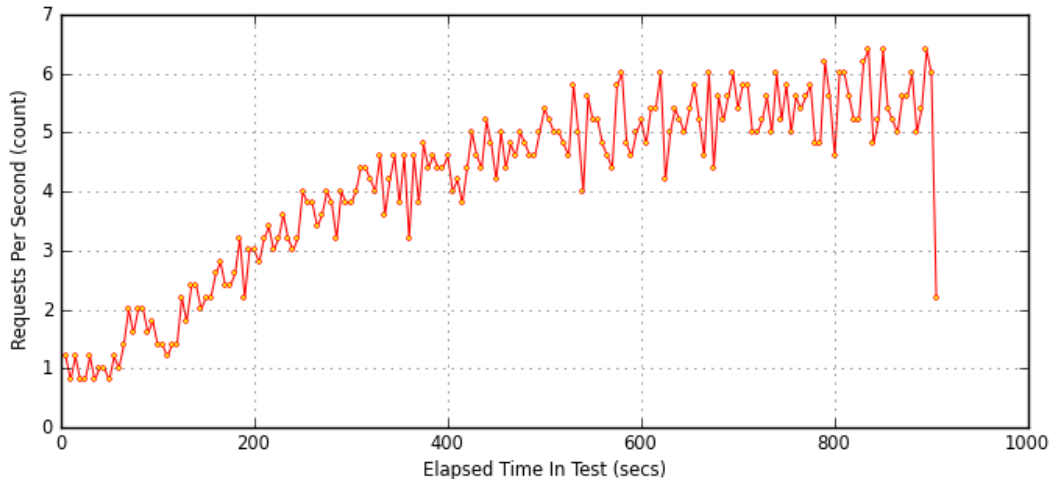
(all samples generated by multi-mechanize using matplotlib)

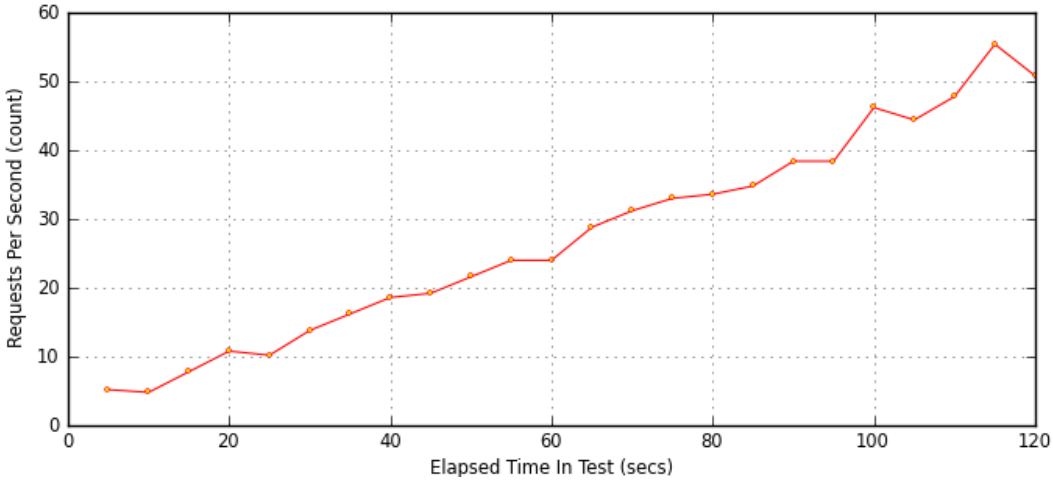



90pct
80pct
Avg
Response Time (secs)
Elapsed Time In Test (secs)


Response Time (secs)
Elapsed Time In Test (secs)


Response Time (secs)
Elapsed Time In Test (secs)





2.5 Database Storage for Test Result Data

Test data and results can be stored in a database when a test run completes. To enable database storage, you must add a `results_database` option to your `config.cfg`, which defines the database connection string.

2.5.1 Example config setting

```
results_database: sqlite:///results.db
```

2.5.2 Requirements

- database storage requires sqlalchemy

2.5.3 Example connection strings

Several database back-ends are available to choose from:

SQLite `sqlite:///dbname`

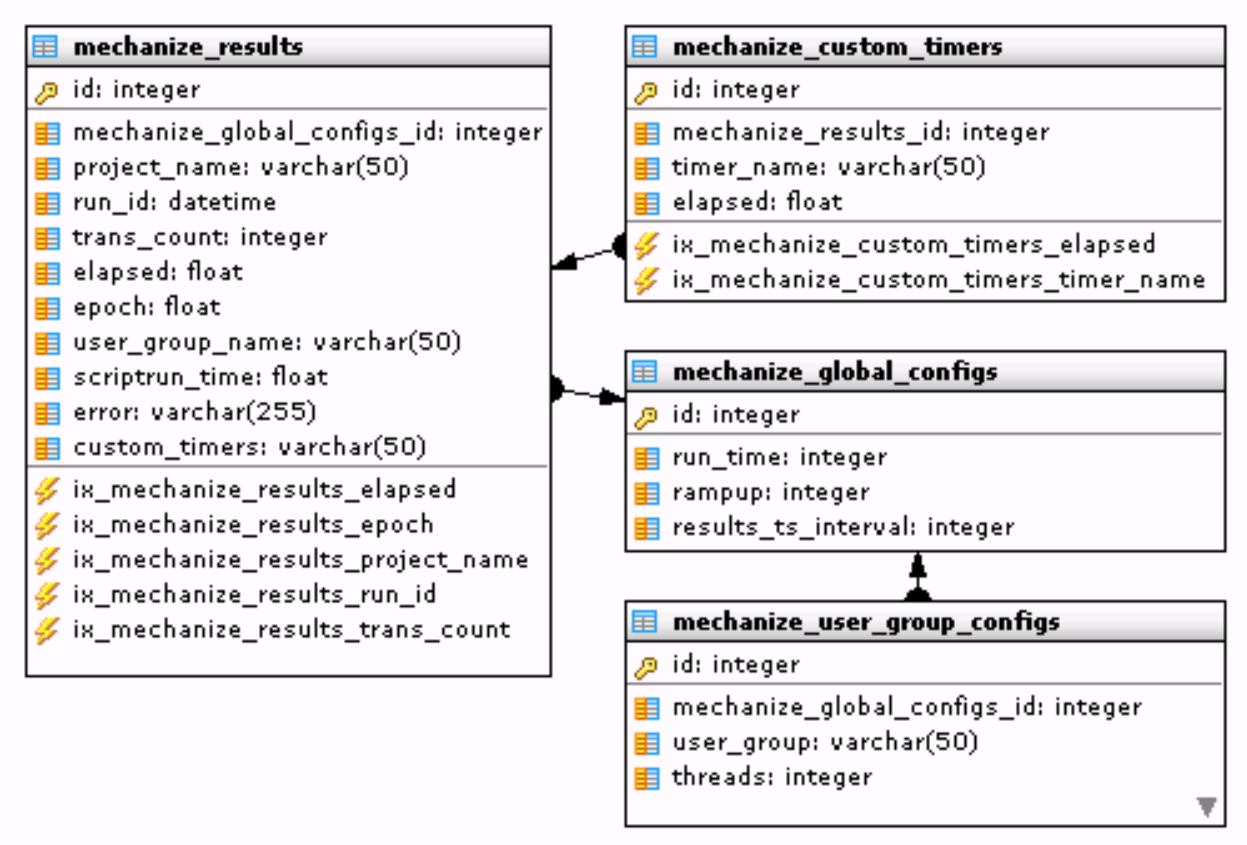
MySQL `mysql://user:password@localhost/dbname`

PostgreSQL `postgresql://user:password@host:port/dbname`

MS SQL Server `mssql://mydsn`

- SQLite is supported natively by Python, so there is no installation or configuration necessary.
- The results database is created automatically on first use, no need to run your own DDL code.

2.5.4 Results Database Diagram



2.5.5 Sample DB Query Code

Here is some sample code for retrieving results from the database via sqlalchemy:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from multimechanize.resultsloader import ResultRow
from multimechanize.resultsloader import GlobalConfig
from multimechanize.resultsloader import UserGroupConfig
from multimechanize.resultsloader import TimerRow
```

```
engine = create_engine('sqlite:///results.db')
session = sessionmaker(bind=engine)
current = session()

for rr in current.query(ResultRow).order_by(ResultRow.trans_count):
    print rr
    print rr.global_config
    print rr.global_config.user_group_configs
    print rr.timers
```

2.6 Development

Multi-Mechanize is Free Open Source Software. Contributors are welcome!

- The Free Software Definition
- The Open Source Definition

2.6.1 GitHub

Multi-Mechanize development is hosted at GitHub and uses Git version control system:

- https://github.com/cgoldberg/multi-mechanize

2.6.2 Dev Communication

- IRC: Freenode `#multimech` channel
- Mailing List: Google Group

2.6.3 Issues

If you have a feature request, suggestion, or bug report, please open a new issue on GitHub.

2.6.4 Patches

To submit a patch, please send a Pull Request on GitHub.

2.6.5 Contributors

Thanks for contributing code to Multi-Mechanize:

- Corey Goldberg
- Brian Knox
- Daniel Sutcliffe
- Wes Winham
- Ali-Akber Saifee

- Richard Leland

2.7 Changelog

2.7.1 Version 1.?.?, 201?-??-??

- bug fixes
- tox config
- travis-ci config

2.7.2 Version 1.2.0, 2012-02-07

- initial release on PyPI (http://pypi.python.org/pypi/multi-mechanize)
- development moved to GitHub (http://github.com/cgoldberg/multi-mechanize)
- using Git for version control (pull requests welcome)
- new entry-point scripts (`multimech-run`, `multimech-newproject`)
- new version numbering format
- new setup and packaging
- new documentation (sphinx generated)
- results re-processing (`-r`` | –results`` command line option)
- specify bind address in rpc-server mode (`-b`` | –bind-addr`` command line option)
- specify project directory location (`-d`` | –directory`` command line option)

Prior to version 1.2.0, Multi-Mechanize used a different version numbering format and was not packaged on PyPI.

2.7.3 Version 1.011, Jun 2011

- v1.011 zip

2.7.4 Version 1.010, Aug 2010

- v1.010 zip

2.7.5 Version 1.009, Mar 2010

- v1.009 zip

2.7.6 Version 1.008, Mar 2010

- v1.008 zip

2.7.7 Version 1.007, Feb 2010

- v1.007 zip

2.7.8 Version 1.006, Feb 2010

- v1.006 zip

2.7.9 Version 1.005, Feb 2010

- v1.005 zip

2.7.10 Version 1.004, Feb 2010

- v1.004 zip

2.7.11 Version 1.003, Jan 2010

- v1.003 zip

2.7.12 Version 1.002, Jan 2010

- v1.002 zip

2.7.13 Version 1.001, Jan 2010

- v1.001 zip

CHAPTER 3

Discussion / Help / Updates

- IRC: Freenode `#multimech` channel
- Mailing List: Google Group
- Twitter: twitter.com/multimechanize

CHAPTER 4

Install / Setup

Multi-Mechanize can be installed from PyPI using pip:

```
pip install -U multi-mechanize
```

... or download the source distribution from PyPI, unarchive, and run:

```
python setup.py install
```

(for more setup and installation instructions, see *Detailed Install and Setup*)

CHAPTER 5

Usage Instructions

5.1 Create a Project

Create a new test project with `multimech-newproject`:

```
$ multimech-newproject my_project
```

Each test project contains the following:

- `config.cfg`: configuration file. set your test options here.
- `test_scripts/`: directory for virtual user scripts. add your test scripts here.
- `results/`: directory for results storage. a timestamped directory is created for each test run, containing the results report.

`multimech-newproject` will create a mock project, using a single script that generates random timer data. Check it out for a basic example.

5.2 Run a Project

Run a test project with `multimech-run`:

```
$ multimech-run my_project
```

- for test configuration options, see *Configuration*
- a timestamped `results` directory is created for each test run, containing the results report.

CHAPTER 6

Test Scripts

6.1 Virtual User Scripting

- written in Python
- test scripts simulate virtual user activity against a site/service/api
- scripts define user transactions
- for help developing scripts, see *Scripting Guide*

6.2 Examples

HTTP GETs using Requests:

```
import requests

class Transaction(object):
    def run(self):
        r = requests.get('https://github.com/timeline.json')
        r.raw.read()
```

HTTP GETs using Mechanize (with timer and assertions):

```
import mechanize
import time

class Transaction(object):
    def run(self):
        br = mechanize.Browser()
        br.set_handle_robots(False)

        start_timer = time.time()
        resp = br.open('http://www.example.com/')
```

```
        resp.read()
        latency = time.time() - start_timer

        self.custom_timers['Example_Homepage'] = latency

        assert (resp.code == 200)
        assert ('Example Web Page' in resp.get_data())
```