Locust

Release 1.0b2

May 01, 2020

Contents

Everything you need to know about Locust

About locust

Locust is a scalable load testing framework written in Python

- **Website**: https://locust.io/
- **Source code**: https://github.com/locustio/locust
- **Twitter**: @locustio

CHAPTER 1

Getting started

1.1 What is Locust?

Locust is an easy-to-use, distributed, user load testing tool. It is intended for load-testing web sites (or other systems) and figuring out how many concurrent users a system can handle.

The idea is that during a test, a swarm of locust users will attack your website. The behavior of each user is defined by you using Python code, and the swarming process is monitored from a web UI in real-time. This will help you battle test and identify bottlenecks in your code before letting real users in.

Locust is completely event-based, and therefore it's possible to support thousands of concurrent users on a single machine. In contrast to many other event-based apps it doesn't use callbacks. Instead it uses light-weight processes, through gevent. Each locust swarming your site is actually running inside its own process (or greenlet, to be correct). This allows you to write very expressive scenarios in Python without complicating your code with callbacks.

1.1.1 Features

- **Write user test scenarios in plain-old Python**

  No need for clunky UIs or bloated XML—just code as you normally would. Based on coroutines instead of callbacks, your code looks and behaves like normal, blocking Python code.

- **Distributed & Scalable - supports hundreds of thousands of users**

  Locust supports running load tests distributed over multiple machines. Being event-based, even one Locust node can handle thousands of users in a single process. Part of the reason behind this is that even if you simulate that many users, not all are actively hitting your system. Often, users are idle figuring out what to do next. Requests per second != number of users online.

- **Web-based UI**

  Locust has a neat HTML+JS user interface that shows relevant test details in real-time. And since the UI is web-based, it's cross-platform and easily extendable.

- **Can test any system**

Even though Locust is web-oriented, it can be used to test almost any system. Just write a client for what ever you wish to test and swarm it with locusts! It's super easy!

- **Hackable**

  Locust is small and very hackable and we intend to keep it that way. All heavy-lifting of evented I/O and coroutines are delegated to gevent. The brittleness of alternative testing tools was the reason we created Locust.

1.1.2 Background

Locust was created because we were fed up with existing solutions. None of them are solving the right problem and to me, they are missing the point. We've tried both Apache JMeter and Tsung. Both tools are quite OK to use; we've used the former many times benchmarking stuff at work. JMeter comes with a UI, which you might think for a second is a good thing. But you soon realize it's a PITA to "code" your testing scenarios through some point-and-click interface. Secondly, JMeter is thread-bound. This means for every user you want to simulate, you need a separate thread. Needless to say, benchmarking thousands of users on a single machine just isn't feasible.

Tsung, on the other hand, does not have these thread issues as it's written in Erlang. It can make use of the light-weight processes offered by BEAM itself and happily scale up. But when it comes to defining the test scenarios, Tsung is as limited as JMeter. It offers an XML-based DSL to define how a user should behave when testing. I guess you can imagine the horror of "coding" this. Displaying any sorts of graphs or reports when completed requires you to post-process the log files generated from the test. Only then can you get an understanding of how the test went.

Anyway, we've tried to address these issues when creating Locust. Hopefully none of the above pain points should exist.

I guess you could say we're really just trying to scratch our own itch here. We hope others will find it as useful as we do.

1.1.3 Authors

- Jonatan Heyman (@jonatanheyman on Twitter)
- Carl Byström (@cgbystrom on Twitter)
- Joakim Hamrén (@Jahaaja on Twitter)
- Hugo Heyman (@hugoheyman on Twitter)

1.1.4 License

Open source licensed under the MIT license (see LICENSE file for details).

1.2 Installation

Locust is available on PyPI and can be installed with pip.

```
$ pip3 install locust
```

If you want the bleeding edge version, you can use pip to install directly from our Git repository. For example, to install the master branch using Python 3:

```
$ pip3 install -e git://github.com/locustio/locust.git@master#egg=locustio
```

Once Locust is installed, a **locust** command should be available in your shell. (if you're not using virtualenv—which you should—make sure your python script directory is on your path).

To see available options, run:

```
$ locust --help
```

1.2.1 Supported Python Versions

Locust is supported on Python 3.6, 3.7 and 3.8.

1.2.2 Installing Locust on Windows

On Windows, running `pip install locustio` *should* work.

However, if it doesn't, chances are that it can be fixed by first installing the pre built binary packages for pyzmq, gevent and greenlet.

You can find an unofficial collection of pre built python packages for windows here: http://www.lfd.uci.edu/~gohlke/pythonlibs/

When you've downloaded a pre-built `.whl` file, you can install it with:

```
$ pip install name-of-file.whl
```

Once you've done that you should be able to just `pip install locustio`.

Note: Running Locust on Windows should work fine for developing and testing your load testing scripts. However, when running large scale tests, it's recommended that you do that on Linux machines, since gevent's performance under Windows is poor.

1.2.3 Installing Locust on macOS

Make sure you have a working installation of Python 3.6 or higher and follow the above instructions. Homebrew can be used to install Python on macOS.

1.2.4 Increasing Maximum Number of Open Files Limit

Every HTTP connection on a machine opens a new file (technically a file descriptor). Operating systems may set a low limit for the maximum number of files that can be open. If the limit is less than the number of simulated users in a test, failures will occur.

Increase the operating system's default maximum number of files limit to a number higher than the number of simulated users you'll want to run. How to do this depends on the operating system in use.

1.3 Quick start

1.3.1 Example locustfile.py

When using Locust you define the behaviour of users in Python code, and then you have the ability to simulate any number of those users while gathering request statistic. The entrypoint for defining the user behaviour is the *locustfile.py*.

Note: The `locustfile.py` is a normal Python file that will get imported by Locust. Within it you can import modules just as you would in any python code.

The file can be named something else and specified with the *-f* flag to the `locust` command.

Below is a quick little example of a simple **locustfile.py**:

```
import random
from locust import HttpUser, task, between

class WebsiteUser(HttpUser):
    wait_time = between(5, 9)

    @task(2)
    def index(self):
        self.client.get("/")
        self.client.get("/ajax-notifications/")

    @task(1)
    def view_post(self):
        post_id = random.randint(1, 10000)
        self.client.get("/post?id=%i" % post_id, name="/post?id=[post-id]")

    def on_start(self):
        """ on_start is called when a User starts before any task is scheduled """
        self.login()

    def login(self):
        self.client.post("/login", {"username":"ellen_key", "password":"education"})
```

1.3.1.1 Let's break it down:

```
class WebsiteUser(HttpUser):
```

Here we define a class for the users that we will be simulating. It inherits from *`HttpUser`* which gives each user a `client` attribute, which is an instance of *`HttpSession`*, that can be used to make HTTP requests to the target system that we want to load test. When a test starts, locust will create an instance of this class for every user that it simulates, and each of these users will start running within their own green gevent thread.

```
wait_time = between(5, 9)
```

Our class defines a `wait_time` function that will make the simulated users wait between 5 and 9 seconds after each task is executed.

```
@task(2)
def index(self):
    self.client.get("/")
    self.client.get("/ajax-notifications/")

@task(1)
def view_post(self):
    ...
```

We've also declared two tasks by decorating two methods with `@task` and given them different weights (2 and 1). When a simulated user of this type runs it'll pick one of either `index` or `view_post` - with twice the chance of picking `index` - call that method and then pick a duration uniformly between 5 and 9 and just sleep for that duration. After it's wait time it'll pick a new task and keep repeating that.

```
def view_post(self):
    post_id = random.randint(1, 10000)
    self.client.get("/post?id=%i" % post_id, name="/post?id=[post-id]")
```

In the `view_post` task we load a dynamic URL by using a query parameter that is a number picked at random between 1 and 10000. In order to not get 10k entries in Locust's statistics - since the stats is grouped on the URL - we use the *name parameter* to group all those requests under an entry named `"/post?id=[post-id]"` instead.

```
def on_start(self):
```

Additionally we've declared a *on_start* method. A method with this name will be called for each simulated user when they start. For more info see *on_start and on_stop methods*.

1.3.2 Start Locust

To run Locust with the above Locust file, if it was named *locustfile.py* and located in the current working directory, we could run:

```
$ locust
```

If the Locust file is located under a subdirectory and/or named different than *locustfile.py*, specify it using `-f`:

```
$ locust -f locust_files/my_locust_file.py
```

To run Locust distributed across multiple processes we would start a master process by specifying `--master`:

```
$ locust -f locust_files/my_locust_file.py --master
```

and then we would start an arbitrary number of worker processes:

```
$ locust -f locust_files/my_locust_file.py --worker
```

If we want to run Locust distributed on multiple machines we would also have to specify the master host when starting the workers (this is not needed when running Locust distributed on a single machine, since the master host defaults to 127.0.0.1):

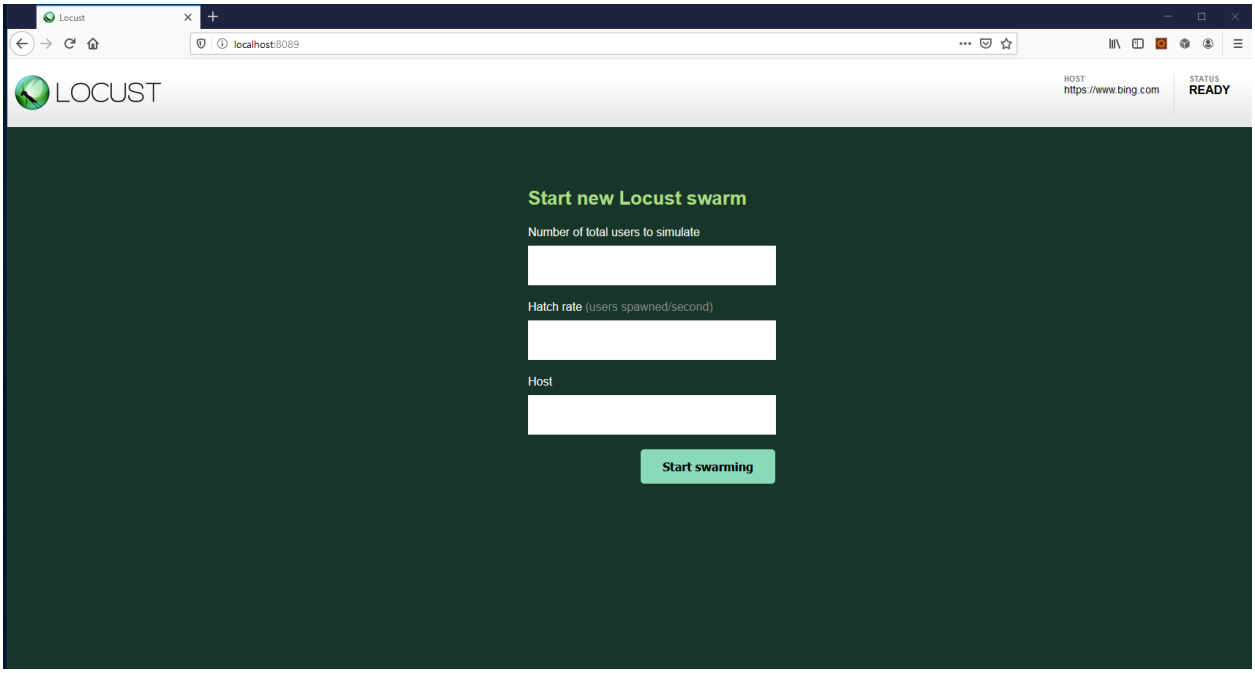
```
$ locust -f locust_files/my_locust_file.py --worker --master-host=192.168.0.100
```

Parameters can also be set through *environment variables*, or in a *config file*.

Note: To see all available options type: `locust --help`

1.3.3 Open up Locust's web interface

Once you've started Locust using one of the above command lines, you should open up a browser and point it to http://127.0.0.1:8089 (if you are running Locust locally). Then you should be greeted with something like this:



1.3.4 Locust Command Line Interface & Configuration

For a full list of available command line options see *Configuration*.

CHAPTER 2

Writing Locust tests

2.1 Writing a locustfile

A locustfile is a normal python file. The only requirement is that it declares at least one class — let's call it the user class — that inherits from the class *User*.

2.1.1 The User class

A user class represents one user (or a swarming locust if you will). Locust will spawn (hatch) one instance of the User class for each user that is being simulated. There are a few attributes that a User class should typically define.

2.1.1.1 The *wait_time* attribute

In addition to the *task_set* attribute, one should also declare a *wait_time* method. It's used to determine for how long a simulated user will wait between executing tasks. Locust comes with a few built in functions that return a few common wait_time methods.

The most common one is *between*. It's used to make the simulated users wait a random time between a min and max value after each task execution. Other built in wait time functions are *constant* and *constant_pacing*.

With the following locustfile, each user would wait between 5 and 15 seconds between tasks:

```
from locust import User, TaskSet, task, between

class MyTaskSet(TaskSet):
    @task
    def my_task(self):
        print("executing my_task")

class MyUser(User):
    tasks = [MyTaskSet]
    wait_time = between(5, 15)
```

The wait_time method should return a number of seconds (or fraction of a second) and can also be declared on a TaskSet class, in which case it will only be used for that TaskSet.

It's also possible to declare your own wait_time method directly on a User or TaskSet class. The following User class would start sleeping for one second and then one, two, three, etc.

```
class MyLocust(Locust):
    task_set = MyTaskSet
    last_wait_time = 0

    def wait_time(self):
        self.last_wait_time += 1
        return self.last_wait_time
```

2.1.1.2 The *weight* attribute

If more than one user class exists in the file, and no user classes are specified on the command line, Locust will spawn an equal number of each of the user classes. You can also specify which of the user classes to use from the same locustfile by passing them as command line arguments:

```
$ locust -f locust_file.py WebUser MobileUser
```

If you wish to simulate more users of a certain type you can set a weight attribute on those classes. Say for example, web users are three times more likely than mobile users:

```
class WebUser(User):
    weight = 3
    ...

class MobileUser(User):
    weight = 1
    ...
```

2.1.1.3 The *host* attribute

The host attribute is a URL prefix (i.e. "http://google.com") to the host that is to be loaded. Usually, this is specified in Locust's web UI or on the command line, using the `--host` option, when locust is started.

If one declares a host attribute in the user class, it will be used in the case when no `--host` is specified on the command line or in the web request.

2.1.1.4 The *tasks* attribute

A User class can have tasks declared as methods under it using the `@task` decorator, but one can also specify tasks using the *tasks* attribute which is described in more details *below*.

2.1.2 Tasks

When a load test is started, an instance of a User class will be created for each simulated user and they will start running within their own green thread. When these users run they pick tasks that they execute, sleeps for awhile, and then picks a new task and so on.

The tasks are normal python callables and — if we were load-testing an auction website — they could do stuff like "loading the start page", "searching for some product", "making a bid", etc.

2.1.2.1 Declaring tasks

The typical way of declaring tasks for a User class (or a TaskSet) it to use the *task* decorator.

Here is an example:

```
from locust import User, task, constant

class MyUser(User):
    wait_time = constant(1)

    @task
    def my_task(self):
        print("User instance (%r) executing my_task" % self)
```

@task takes an optional weight argument that can be used to specify the task's execution ratio. In the following example *task2* will have twice the chance of being picked as *task1*:

```
from locust import User, task, between

class MyUser(User):
    wait_time = between(5, 15)

    @task(3)
    def task1(self):
        pass

    @task(6)
    def task2(self):
        pass
```

2.1.2.2 tasks attribute

Using the @task decorator to declare tasks is a convenience, and usually the best way to do it. However, it's also possible to define the tasks of a User or TaskSet by setting the *tasks* attribute (using the @task decorator will actually just populate the *tasks* attribute).

The *tasks* attribute is either a list of Task, or a <*Task : int*> dict, where Task is either a python callable or a TaskSet class (more on that below). If the task is a normal python function they receive a single argument which is the User instance that is executing the task.

Here is an example of a User task declared as a normal python function:

```
from locust import User, constant

def my_task(l):
    pass

class MyUser(User):
    tasks = [my_task]
    wait_time = constant(1)
```

If the tasks attribute is specified as a list, each time a task is to be performed, it will be randomly chosen from the *tasks* attribute. If however, *tasks* is a dict — with callables as keys and ints as values — the task that is to be executed will be chosen at random but with the int as ratio. So with a tasks that looks like this:

```
{my_task: 3, another_task: 1}
```

my_task would be 3 times more likely to be executed than *another_task*.

Internally the above dict will actually be expanded into a list (and the `tasks` attribute is updated) that looks like this:

```
[my_task, my_task, my_task, another_task]
```

and then Python's `random.choice()` is used pick tasks from the list.

2.1.3 TaskSet class

Since real websites are usually built up in an hierarchical way, with multiple sub-sections, locust has the TaskSet class. A locust task can not only be a Python callable, but also a TaskSet class. A TaskSet is a collection of locust tasks that will be executed much like the tasks declared directly on a User class, with the user sleeping in between task executions. Here's a short example of a locustfile that has a TaskSet:

```
from locust import User, TaskSet, between

class ForumSection(TaskSet):
    @task(10)
    def view_thread(self):
        pass

    @task(1)
    def create_thread(self):
        pass

    @task(1)
    def stop(self):
        self.interrupt()

class LoggedInUser(User):
    wait_time = between(5, 120)
    tasks = {ForumSection:2}

    @task
    def index_page(self):
        pass
```

A TaskSet can also be inlined directly under a User/TaskSet class using the @task decorator:

```
class MyUser(User):
    @task(1)
    class MyTaskSet(TaskSet):
        ...
```

The tasks of a TaskSet class can be other TaskSet classes, allowing them to be nested any number of levels. This allows us to define a behaviour that simulates users in a more realistic way. For example we could define TaskSets with the following structure:

```
- Main user behaviour
  - Index page
  - Forum page
    - Read thread
      - Reply
```

(continues on next page)

(continued from previous page)

```
    - New thread
    - View next page
  - Browse categories
    - Watch movie
    - Filter movies
  - About page
```

When a running User thread picks a TaskSet class for execution an instance of this class will be created and execution will then go into this TaskSet. What happens then is that one of the TaskSet's tasks will be picked and executed, and then the thread will sleep for a duration specified by the User's wait_time function (unless a `wait_time` function has been declared directly on the TaskSet class, in which case it'll use that function instead), then pick a new task from the TaskSet's tasks, wait again, and so on.

2.1.3.1 Interrupting a TaskSet

One important thing to know about TaskSets is that they will never stop executing their tasks, and hand over execution back to their parent User/TaskSet, by themselves. This has to be done by the developer by calling the *`TaskSet.interrupt()`* method.

interrupt(*self*, *reschedule=True*)

Interrupt the TaskSet and hand over execution control back to the parent TaskSet.

If *reschedule* is True (default), the parent User will immediately re-schedule, and execute, a new task.

This method should not be called by the root TaskSet (the one that is immediately, attached to the User class's *task_set* attribute), but rather in nested TaskSet classes further down the hierarchy.

In the following example, if we didn't have the stop task that calls `self.interrupt()`, the simulated user would never stop running tasks from the Forum taskset once it has went into it:

```
class RegisteredUser(User):
    @task
    class Forum(TaskSet):
        @task(5)
        def view_thread(self):
            pass

        @task(1)
        def stop(self):
            self.interrupt()

    @task
    def frontpage(self):
        pass
```

Using the interrupt function, we can — together with task weighting — define how likely it is that a simulated user leaves the forum.

2.1.3.2 Differences between tasks in TaskSet and User classes

One difference for tasks residing under a TaskSet, compared to tasks residing directly under a User, is that the argument that they are passed when executed (`self` for tasks declared as methods with the *`@task`* decorator) is a reference to the TaskSet instance, instead of the User instance. The User instance can be accessed from within a TaskSet instance through the *`TaskSet.user`*. TaskSets also contains a convenience *`client`* attribute that refers to the client attribute on the User instance.

2.1.3.3 Referencing the User instance, or the parent TaskSet instance

A TaskSet instance will have the attribute *locust* point to its User instance, and the attribute *parent* point to its parent TaskSet instance.

2.1.4 SequentialTaskSet class

SequentialTaskSet is a TaskSet but its tasks will be executed in the order that they are declared. Weights are ignored for tasks on a SequentialTaskSet class. Ofcourse you can also nest SequentialTaskSet within TaskSet and vice versa.

```
def function_task(taskset):
    pass

class SequenceOfTasks(SequentialTaskSet):
    @task
    def first_task(self):
        pass

    tasks = [functon_task]

    @task
    def second_task(self):
        pass

    @task
    def third_task(self):
        pass
```

In the above example, the tasks are executed in the order of declaration:

1. `first_task`
2. `function_task`
3. `second_task`
4. `third_task`

and then it will start over at `first_task` again.

2.1.5 on_start and on_stop methods

User and TaskSet classes can declare an *on_start* method and/or *on_stop* method. A User will call it's *on_start* method when it starts running, and it's *on_stop* method when it stops running. For a TaskSet, the *on_start* method is called when a simulated user starts executing that TaskSet, and *on_stop* is called when the simulated user stops executing that TaskSet (when *interrupt()* is called, or the user is killed).

2.1.6 test_start and test_stop events

If you need to run some code at the start or stop of a load test, you should use the *test_start* and *test_stop* events. You can set up listeners for these events at the module level of your locustfile:

```
from locust import events

@events.test_start.add_listener
def on_test_start(**kwargs):
    print("A new test is starting")

@events.test_stop.add_listener
def on_test_stop(**kwargs):
    print("A new test is ending")
```

When running Locust distributed the `test_start` and `test_stop` events will only be fired in the master node.

2.1.7 Making HTTP requests

So far, we've only covered the task scheduling part of a User. In order to actually load test a system we need to make HTTP requests. To help us do this, the *`HttpLocust`* class exists. When using this class, each instance gets a `client` attribute which will be an instance of *`HttpSession`* which can be used to make HTTP requests.

class HttpUser(**args*, ***kwargs*)

Represents an HTTP "user" which is to be hatched and attack the system that is to be load tested.

The behaviour of this user is defined by its tasks. Tasks can be declared either directly on the class by using the *`@task decorator`* on methods, or by setting the *`tasks attribute`*.

This class creates a *client* attribute on instantiation which is an HTTP client with support for keeping a user session between requests.

client = None

Instance of HttpSession that is created upon instantiation of Locust. The client supports cookies, and therefore keeps the session between HTTP requests.

When inheriting from the HttpUser class, we can use its client attribute to make HTTP requests against the server. Here is an example of a locust file that can be used to load test a site with two URLs; **/** and **/about/**:

```
from locust import HttpUser, task, between

class MyUser(HttpUser):
    wait_time = between(5, 15)

    @task(2)
    def index(self):
        self.client.get("/")

    @task(1)
    def about(self):
        self.client.get("/about/")
```

Using the above User class, each simulated user will wait between 5 and 15 seconds between the requests, and **/** will be requested twice as much as **/about/**.

2.1.7.1 Using the HTTP client

Each instance of HttpUser has an instance of *`HttpSession`* in the *client* attribute. The HttpSession class is actually a subclass of `requests.Session` and can be used to make HTTP requests, that will be reported to User's statistics, using the *`get`*, *`post`*, *`put`*, *`delete`*, *`head`*, *`patch`* and *`options`* methods. The HttpSession instance will preserve cookies between requests so that it can be used to log in to websites and keep a session between requests. The client

attribute can also be referenced from the User instance's TaskSet instances so that it's easy to retrieve the client and make HTTP requests from within your tasks.

Here's a simple example that makes a GET request to the */about* path (in this case we assume *self* is an instance of a *TaskSet* or *HttpUser* class:

```
response = self.client.get("/about")
print("Response status code:", response.status_code)
print("Response content:", response.text)
```

And here's an example making a POST request:

```
response = self.client.post("/login", {"username":"testuser", "password":"secret"})
```

2.1.7.2 Safe mode

The HTTP client is configured to run in safe_mode. What this does is that any request that fails due to a connection error, timeout, or similar will not raise an exception, but rather return an empty dummy Response object. The request will be reported as a failure in User's statistics. The returned dummy Response's *content* attribute will be set to None, and its *status_code* will be 0.

2.1.7.3 Manually controlling if a request should be considered successful or a failure

By default, requests are marked as failed requests unless the HTTP response code is OK (<400). Most of the time, this default is what you want. Sometimes however—for example when testing a URL endpoint that you expect to return 404, or testing a badly designed system that might return *200 OK* even though an error occurred—there's a need for manually controlling if locust should consider a request as a success or a failure.

One can mark requests as failed, even when the response code is OK, by using the *catch_response* argument and a with statement:

```
with self.client.get("/", catch_response=True) as response:
    if response.content != b"Success":
        response.failure("Got wrong response")
```

Just as one can mark requests with OK response codes as failures, one can also use **catch_response** argument together with a *with* statement to make requests that resulted in an HTTP error code still be reported as a success in the statistics:

```
with self.client.get("/does_not_exist/", catch_response=True) as response:
    if response.status_code == 404:
        response.success()
```

2.1.7.4 Grouping requests to URLs with dynamic parameters

It's very common for websites to have pages whose URLs contain some kind of dynamic parameter(s). Often it makes sense to group these URLs together in User's statistics. This can be done by passing a *name* argument to the *HttpSession's* different request methods.

Example:

```
# Statistics for these requests will be grouped under: /blog/?id=[id]
for i in range(10):
    self.client.get("/blog?id=%i" % i, name="/blog?id=[id]")
```

2.1.7.5 HTTP Proxy settings

To improve performance, we configure requests to not look for HTTP proxy settings in the environment by setting requests.Session's trust_env attribute to `False`. If you don't want this you can manually set `locust_instance.client.trust_env` to `True`. For further details, refer to the documentation of requests.

2.1.8 How to structure your test code

It's important to remember that the locustfile.py is just an ordinary Python module that is imported by Locust. From this module you're free to import other python code just as you normally would in any Python program. The current working directory is automatically added to python's `sys.path`, so any python file/module/packages that resides in the working directory can be imported using the python `import` statement.

For small tests, keeping all of the test code in a single `locustfile.py` should work fine, but for larger test suites, you'll probably want to split the code into multiple files and directories.

How you structure the test source code is ofcourse entirely up to you, but we recommend that you follow Python best practices. Here's an example file structure of an imaginary Locust project:

- Project root
  - `common/`
    - `__init__.py`
    - `auth.py`
    - `config.py`
  - `locustfile.py`
  - `requirements.txt` (External Python dependencies is often kept in a requirements.txt)

A project with multiple different locustfiles could also keep them in a separate subdirectory:

- Project root
  - `common/`
    - `__init__.py`
    - `auth.py`
    - `config.py`
  - `locustfiles/`
    - `api.py`
    - `website.py`
  - `requirements.txt`

With any of the above project structure, your locustfile can import common libraries using:

```
import common.auth
```

CHAPTER 3

Running your Locust tests

3.1 Configuration

3.1.1 Command Line Options

The most straight forward way to Configure how Locust is run is through command line arguments.

```
$ locust --help
```

```
Usage: locust [OPTIONS] [UserClass ...]

Common options:
  -h, --help            show this help message and exit
  -f LOCUSTFILE, --locustfile LOCUSTFILE
                        Python module file to import, e.g. '../other.py'.
                        Default: locustfile
  --config CONFIG       Config file path
  -H HOST, --host HOST  Host to load test in the following format:
                        http://10.21.32.33
  -u NUM_USERS, --users NUM_USERS
                        Number of concurrent Locust users. Only used together
                        with --headless
  -r HATCH_RATE, --hatch-rate HATCH_RATE
                        The rate per second in which users are spawned. Only
                        used together with --headless
  -t RUN_TIME, --run-time RUN_TIME
                        Stop after the specified amount of time, e.g. (300s,
                        20m, 3h, 1h30m, etc.). Only used together with
                        --headless
  -l, --list            Show list of possible User classes and exit

Web UI options:
  --web-host WEB_HOST   Host to bind the web interface to. Defaults to '*'
```

(continues on next page)

(continued from previous page)

```
                        (all interfaces)
  --web-port WEB_PORT, -P WEB_PORT
                        Port on which to run web host
  --headless            Disable the web interface, and instead start the load
                        test immediately. Requires -u and -t to be specified.
  --web-auth WEB_AUTH   Turn on Basic Auth for the web interface. Should be
                        supplied in the following format: username:password
  --tls-cert TLS_CERT   Optional path to TLS certificate to use to serve over
                        HTTPS
  --tls-key TLS_KEY     Optional path to TLS private key to use to serve over
                        HTTPS

Master options:
  Options for running a Locust Master node when running Locust distributed. A Master␣
↪node need Worker nodes that connect to it before it can run load tests.

  --master              Set locust to run in distributed mode with this
                        process as master
  --master-bind-host MASTER_BIND_HOST
                        Interfaces (hostname, ip) that locust master should
                        bind to. Only used when running with --master.
                        Defaults to * (all available interfaces).
  --master-bind-port MASTER_BIND_PORT
                        Port that locust master should bind to. Only used when
                        running with --master. Defaults to 5557.
  --expect-workers EXPECT_WORKERS
                        How many workers master should expect to connect
                        before starting the test (only when --headless used).

Worker options:

  Options for running a Locust Worker node when running Locust distributed.
  Only the LOCUSTFILE (-f option) need to be specified when starting a Worker, since␣
↪other options such as -u, -r, -t are specified on the Master node.

  --worker              Set locust to run in distributed mode with this
                        process as worker
  --master-host MASTER_HOST
                        Host or IP address of locust master for distributed
                        load testing. Only used when running with --worker.
                        Defaults to 127.0.0.1.
  --master-port MASTER_PORT
                        The port to connect to that is used by the locust
                        master for distributed load testing. Only used when
                        running with --worker. Defaults to 5557.

Request statistics options:
  --csv CSV_PREFIX      Store current request stats to files in CSV format.
                        Setting this option will generate three files:
                        [CSV_PREFIX]_stats.csv, [CSV_PREFIX]_stats_history.csv
                        and [CSV_PREFIX]_failures.csv
  --csv-full-history    Store each stats entry in CSV format to
                        _stats_history.csv file
  --print-stats         Print stats in the console
  --only-summary        Only print the summary stats
  --reset-stats         Reset statistics once hatching has been completed.
                        Should be set on both master and workers when running
```

(continues on next page)

(continued from previous page)

```
                        in distributed mode

Logging options:
  --skip-log-setup      Disable Locust's logging setup. Instead, the
                        configuration is provided by the Locust test or Python
                        defaults.
  --loglevel LOGLEVEL, -L LOGLEVEL
                        Choose between DEBUG/INFO/WARNING/ERROR/CRITICAL.
                        Default is INFO.
  --logfile LOGFILE     Path to log file. If not set, log will go to
                        stdout/stderr

Step load options:
  --step-load           Enable Step Load mode to monitor how performance
                        metrics varies when user load increases. Requires
                        --step-users and --step-time to be specified.
  --step-users STEP_USERS
                        User count to increase by step in Step Load mode. Only
                        used together with --step-load
  --step-time STEP_TIME
                        Step duration in Step Load mode, e.g. (300s, 20m, 3h,
                        1h30m, etc.). Only used together with --step-load

Other options:
  --show-task-ratio     Print table of the User classes' task execution ratio
  --show-task-ratio-json
                        Print json data of the User classes' task execution
                        ratio
  --version, -V         Show program's version number and exit
  --exit-code-on-error EXIT_CODE_ON_ERROR
                        Sets the process exit code to use when a test result
                        contain any failure or error
  -s STOP_TIMEOUT, --stop-timeout STOP_TIMEOUT
                        Number of seconds to wait for a simulated user to
                        complete any executing task before exiting. Default is
                        to terminate immediately. This parameter only needs to
                        be specified for the master process when running
                        Locust distributed.

User classes:
  UserClass             Optionally specify which User classes that should be
                        used (available User classes can be listed with -l or
                        --list)
```

3.1.2 Environment Variables

Most of the options that can be set through command line arguments can also be set through environment variables. Example:

3.1.3 Configuration File

Any of the options that can be set through command line arguments can also be set by a configuration file in the config file format.

Locust will look for `locust.conf` or `~/.locust.conf` by default, or a file may be specified with the `--config` flag. Parameters passed as environment variables will override the settings from the config file, and command line arguments will override the settings from both environment variables and config file.

Example:

```
$ locust --config=master.conf
```

3.1.4 All available configuration options

Here's a table of all the available configuration options, and their corresponding Environment and config file keys:

Command line	Environment	Config file	Description
-f, --locustfile	LOCUST_LOCUSTFILE	locustfile	Python module file to import,
-H, --host	LOCUST_HOST	host	Host to load test in the followi
-u, --users	LOCUST_USERS	users	Number of concurrent Locust
-r, --hatch-rate	LOCUST_HATCH_RATE	hatch-rate	The rate per second in which u
-t, --run-time	LOCUST_RUN_TIME	run-time	Stop after the specified amoun
--web-host	LOCUST_WEB_HOST	web-host	Host to bind the web interface
--web-port, -P	LOCUST_WEB_PORT	web-port	Port on which to run web host
--headless	LOCUST_HEADLESS	headless	Disable the web interface, and
--web-auth	LOCUST_WEB_AUTH	web-auth	Turn on Basic Auth for the we
--tls-cert	LOCUST_TLS_CERT	tls-cert	Optional path to TLS certifica
--tls-key	LOCUST_TLS_KEY	tls-key	Optional path to TLS private k
--master	LOCUST_MODE_MASTER	master	Set locust to run in distributed
--master-bind-host	LOCUST_MASTER_BIND_HOST	master-bind-host	Interfaces (hostname, ip) that
--master-bind-port	LOCUST_MASTER_BIND_PORT	master-bind-port	Port that locust master should
--expect-workers	LOCUST_EXPECT_WORKERS	expect-workers	How many workers master sho
--worker	LOCUST_MODE_WORKER	worker	Set locust to run in distributed
--master-host	LOCUST_MASTER_NODE_HOST	master-host	Host or IP address of locust m
--master-port	LOCUST_MASTER_NODE_PORT	master-port	The port to connect to that is u
--csv	LOCUST_CSV	csv	Store current request stats to fi
--csv-full-history	LOCUST_CSV_FULL_HISTORY	csv-full-history	Store each stats entry in CSV
--print-stats	LOCUST_PRINT_STATS	print-stats	Print stats in the console
--only-summary	LOCUST_ONLY_SUMMARY	only-summary	Only print the summary stats
--reset-stats	LOCUST_RESET_STATS	reset-stats	Reset statistics once hatching
--skip-log-setup	LOCUST_SKIP_LOG_SETUP	skip-log-setup	Disable Locust's logging setup
--loglevel, -L	LOCUST_LOGLEVEL	loglevel	Choose between DEBUG/INF
--logfile	LOCUST_LOGFILE	logfile	Path to log file. If not set, log
--step-load	LOCUST_STEP_LOAD	step-load	Enable Step Load mode to mo
--step-users	LOCUST_STEP_USERS	step-users	User count to increase by step
--step-time	LOCUST_STEP_TIME	step-time	Step duration in Step Load mo
--exit-code-on-error	LOCUST_EXIT_CODE_ON_ERROR	exit-code-on-error	Sets the process exit code to u
-s, --stop-timeout	LOCUST_STOP_TIMEOUT	stop-timeout	Number of seconds to wait for

3.2 Running Locust distributed

Once a single machine isn't enough to simulate the number of users that you need, Locust supports running load tests distributed across multiple machines.

To do this, you start one instance of Locust in master mode using the `--master` flag. This is the instance that will be running Locust's web interface where you start the test and see live statistics. The master node doesn't simulate any users itself. Instead you have to start one or —most likely—multiple worker Locust nodes using the `--worker` flag, together with the `--master-host` (to specify the IP/hostname of the master node).

A common set up is to run a single master on one machine, and then run **one worker instance per processor core** on the worker machines.

Note: Both the master and each worker machine, must have a copy of the locust test scripts when running Locust distributed.

Note: It's recommended that you start a number of simulated users that are greater than `number of user classes * number of workers` when running Locust distributed.

Otherwise - due to the current implementation - you might end up with a distribution of the User classes that doesn't correspond to the User classes' `weight` attribute. And if the hatch rate is lower than the number of worker nodes, the hatching would occur in "bursts" where all worker node would hatch a single user and then sleep for multiple seconds, hatch another user, sleep and repeat.

3.2.1 Example

To start locust in master mode:

```
locust -f my_locustfile.py --master
```

And then on each worker (replace `192.168.0.14` with IP of the master machine):

```
locust -f my_locustfile.py --worker --master-host=192.168.0.14
```

3.2.2 Options

3.2.2.1 `--master`

Sets locust in master mode. The web interface will run on this node.

3.2.2.2 `--worker`

Sets locust in worker mode.

3.2.2.3 `--master-host=X.X.X.X`

Optionally used together with `--worker` to set the hostname/IP of the master node (defaults to 127.0.0.1)

3.2.2.4 `--master-port=5557`

Optionally used together with `--worker` to set the port number of the master node (defaults to 5557).

3.2.2.5 --master-bind-host=X.X.X.X

Optionally used together with `--master`. Determines what network interface that the master node will bind to. Defaults to * (all available interfaces).

3.2.2.6 --master-bind-port=5557

Optionally used together with `--master`. Determines what network ports that the master node will listen to. Defaults to 5557.

3.2.2.7 --expect-workers=X

Used when starting the master node with `--headless`. The master node will then wait until X worker nodes has connected before the test is started.

3.2.3 Running distributed with Docker

See *Running Locust with Docker*

3.2.4 Running Locust distributed without the web UI

See *Running Locust distributed without the web UI*

3.2.5 Running Locust distributed in Step Load mode

See *Running Locust in Step Load Mode*

3.2.6 Increase Locust's performance

If you're planning to run large-scale load tests you might be interested to use the alternative HTTP client that's shipped with Locust. You can read more about it here: *Increase Locust's performance with a faster HTTP client*

3.3 Running Locust with Docker

The official Docker image is currently found at locustio/locust.

The docker image can be used like this (assuming that the `locustifile.py` exists in the current working directory):

```
docker run -p 8089:8089 -v $PWD:/mnt/locust locustio/locust -f /mnt/locust/locustfile.
↪py
```

3.3.1 Docker Compose

Here's an example Docker Compose file that could ber used to start both a master node, and slave nodes:

```
version: '3'

services:
  master:
    image: locustio/locust
    ports:
     - "8089:8089"
    volumes:
      - ./:/mnt/locust
    command: -f /mnt/locust/locustfile.py --master -H http://master:8089

  worker:
    image: locustio/locust
    volumes:
      - ./:/mnt/locust
    command: -f /mnt/locust/locustfile.py --worker --master-host master
```

The above compose configuration could be used to start a master node and 4 workers using the following command:

```
docker-compose up --scale worker=4
```

3.3.2 Use docker image as a base image

It's very common to have test scripts that rely on third party python packages. In those cases you can use the official Locust docker image as a base image:

```
FROM locustio/locust
RUN pip3 install some-python-package
```

3.4 Running Locust without the web UI

You can run locust without the web UI - for example if you want to run it in some automated flow, like a CI server - by using the `--headless` flag together with `-u` and `-r`:

```
$ locust -f locust_files/my_locust_file.py --headless -u 1000 -r 100
```

`-u` specifies the number of Users to spawn, and `-r` specifies the hatch rate (number of users to spawn per second).

3.4.1 Setting a time limit for the test

If you want to specify the run time for a test, you can do that with `--run-time` or `-t`:

```
$ locust -f --headless -u 1000 -r 100 --run-time 1h30m
```

Locust will shutdown once the time is up.

3.4.2 Allow tasks to finish their iteration on shutdown

By default, locust will stop your tasks immediately. If you want to allow your tasks to finish their iteration, you can use `--stop-timeout <seconds>`.

```
$ locust -f --headless -u 1000 -r 100 --run-time 1h30m --stop-timeout 99
```

3.4.3 Running Locust distributed without the web UI

If you want to *run Locust distributed* without the web UI, you should specify the `--expect-workers` option when starting the master node, to specify the number of worker nodes that are expected to connect. It will then wait until that many worker nodes have connected before starting the test.

3.5 Increase Locust's performance with a faster HTTP client

Locust's default HTTP client uses python-requests. The reason for this is that requests is a very well-maintained python package, that provides a really nice API, that many python developers are familiar with. Therefore, in many cases, we recommend that you use the default *HttpUser* which uses requests. However, if you're planning to run really large scale tests, Locust comes with an alternative HTTP client, *FastHttpUser* which uses geventhttpclient instead of requests. This client is significantly faster, and we've seen 5x-6x performance increases for making HTTP-requests. This does not necessarily mean that the number of users one can simulate per CPU core will automatically increase 5x-6x, since it also depends on what else the load testing script does. However, if your locust scripts are spending most of their CPU time in making HTTP-requests, you are likely to see significant performance gains.

3.5.1 How to use FastHttpUser

Subclass FastHttpUser instead of HttpUser:

```
from locust import task, between
from locust.contrib.fasthttp import FastHttpUser

class MyUser(FastHttpUser):
    wait_time = between(1, 60)

    @task
    def index(self):
        response = self.client.get("/")
```

Note: FastHttpUser uses a whole other HTTP client implementation, with a different API, compared to the default HttpUser that uses python-requests. Therefore FastHttpUser might not work as a drop-in replacement for HttpUser, depending on how the HttpClient is used.

3.5.2 API

3.5.2.1 FastHttpUser class

class FastHttpUser(*environment*)

FastHttpUser uses a different HTTP client (geventhttpclient) compared to HttpUser (python-requests). It's significantly faster, but not as capable.

The behaviour of this user is defined by it's tasks. Tasks can be declared either directly on the class by using the *@task decorator* on the methods, or by setting the *tasks attribute*.

This class creates a *client* attribute on instantiation which is an HTTP client with support for keeping a user session between requests.

connection_timeout = 60.0
Parameter passed to FastHttpSession

insecure = True
Parameter passed to FastHttpSession. Default True, meaning no SSL verification.

max_redirects = 5
Parameter passed to FastHttpSession. Default 5, meaning 4 redirects.

max_retries = 1
Parameter passed to FastHttpSession. Default 1, meaning zero retries.

network_timeout = 60.0
Parameter passed to FastHttpSession

3.5.2.2 FastHttpSession class

class FastHttpSession(*environment: locust.env.Environment*, *base_url: str*, ***kwargs*)

get(*path*, ***kwargs*)
Sends a GET request

head(*path*, ***kwargs*)
Sends a HEAD request

options(*path*, ***kwargs*)
Sends a OPTIONS request

patch(*path*, *data=None*, ***kwargs*)
Sends a POST request

post(*path*, *data=None*, ***kwargs*)
Sends a POST request

put(*path*, *data=None*, ***kwargs*)
Sends a PUT request

request(*method: str*, *path: str*, *name: str = None*, *data: str = None*, *catch_response: bool = False*, *stream: bool = False*, *headers: dict = None*, *auth=None*, *json: dict = None*, *allow_redirects=True*, ***kwargs*)
Send and HTTP request Returns *locust.contrib.fasthttp.FastResponse* object.

Parameters

- **method** – method for the new `Request` object.
- **path** – Path that will be concatenated with the base host URL that has been specified. Can also be a full URL, in which case the full URL will be requested, and the base host is ignored.
- **name** – (optional) An argument that can be specified to use as label in Locust's statistics instead of the URL path. This can be used to group different URL's that are requested into a single entry in Locust's statistics.
- **catch_response** – (optional) Boolean argument that, if set, can be used to make a request return a context manager to work as argument to a with statement. This will allow the request to be marked as a fail based on the content of the response, even if the response

code is ok (2xx). The opposite also works, one can use catch_response to catch a request and then mark it as successful even if the response code was not (i.e 500 or 404).

- **data** – (optional) String/bytes to send in the body of the request.
- **json** – (optional) Dictionary to send in the body of the request. Automatically sets Content-Type and Accept headers to "application/json". Only used if data is not set.
- **headers** – (optional) Dictionary of HTTP Headers to send with the request.
- **auth** – (optional) Auth (username, password) tuple to enable Basic HTTP Auth.
- **stream** – (optional) If set to true the response body will not be consumed immediately and can instead be consumed by accessing the stream attribute on the Response object. Another side effect of setting stream to True is that the time for downloading the response content will not be accounted for in the request time that is reported by Locust.

class FastResponse(*ghc_response*, *request=None*, *sent_request=None*)

content
Unzips if necessary and buffers the received body. Careful with large files!

headers = None
Dict like object containing the response headers

json() → dict
Parses the response as json and returns a dict

text
Returns the text content of the response as a decoded string

CHAPTER 4

Other functionalities

4.1 Running Locust in Step Load Mode

If you want to monitor your service performance with different user load and probe the max tps that can be achieved, you can run Locust with Step Load enabled with `--step-load`:

```
$ locust -f locust_files/my_locust_file.py --step-load
```

4.1.1 Options

4.1.1.1 `--step-load`

Enable Step Load mode to monitor how performance metrics varies when user load increases.

4.1.1.2 `--step-clients`

Client count to increase by step in Step Load mode. Only used together with `--step-load`.

4.1.1.3 `--step-time`

Step duration in Step Load mode, e.g. (300s, 20m, 3h, 1h30m, etc.). Only used together with `--step-load`.

4.1.1.4 Running Locust in step load mode without the web UI

If you want to run Locust in step load mode without the web UI, you can do that with `--step-clients` and `--step-time`:

```
$ locust -f --headless -u 1000 -r 100 --run-time 1h30m --step-load --step-clients 300␣
↪--step-time 20m
```

Locust will swarm the clients by step and shutdown once the time is up.

4.1.1.5 Running Locust distributed in step load mode

If you want to *run Locust distributed* in step load mode, you should specify the `--step-load` option when starting the master node, to swarm locusts by step. It will then show the `--step-clients` option and `--step-time` option in Locust UI.

4.2 Retrieve test statistics in CSV format

You may wish to consume your Locust results via a CSV file. In this case, there are two ways to do this.

First, when running Locust with the web UI, you can retrieve CSV files under the Download Data tab.

Secondly, you can run Locust with a flag which will periodically save trhee CSV files. This is particularly useful if you plan on running Locust in an automated way with the `--headless` flag:

```
$ locust -f examples/basic.py --csv=example --headless -t10m
```

The files will be named `example_stats.csv`, `example_failures.csv` and `example_history.csv` (when using `--csv=example`). The first two files will contain the stats and failures for the whole test run, with a row for every stats entry (URL endpoint) and an aggregated row. The `example_history.csv` will get new rows with the *current* (10 seconds sliding window) stats appanded during the whole test run. By default only the Aggregate row is appended regularly to the history stats, but if Locust is started with the `--csv-full-history` flag, a row for each stats entry (and the Aggregate) is appended every time the stats are written (once every 2 seconds by default).

You can also customize how frequently this is written if you desire faster (or slower) writing:

```
import locust.stats
locust.stats.CSV_STATS_INTERVAL_SEC = 5 # default is 2 seconds
```

4.3 Testing other systems using custom clients

Locust was built with HTTP as its main target. However, it can easily be extended to load test any request/response based system, by writing a custom client that triggers *request_success* and *request_failure* events.

4.3.1 Sample XML-RPC User client

Here is an example of a User class, **XmlRpcUser**, which provides an XML-RPC client, **XmlRpcUser**, and tracks all requests made:

```
import time
from xmlrpc.client import ServerProxy, Fault

from locust import User, task, between


class XmlRpcClient(ServerProxy):
    """
    Simple, sample XML RPC client implementation that wraps xmlrpclib.ServerProxy and
    fires locust events on request_success and request_failure, so that all requests
```

(continues on next page)

(continued from previous page)

```
    gets tracked in locust's statistics.
    """

    _locust_environment = None

    def __getattr__(self, name):
        func = ServerProxy.__getattr__(self, name)
        def wrapper(*args, **kwargs):
            start_time = time.time()
            try:
                result = func(*args, **kwargs)
            except Fault as e:
                total_time = int((time.time() - start_time) * 1000)
                self._locust_environment.events.request_failure.fire(request_type=
↪"xmlrpc", name=name, response_time=total_time, exception=e)
            else:
                total_time = int((time.time() - start_time) * 1000)
                self._locust_environment.events.request_success.fire(request_type=
↪"xmlrpc", name=name, response_time=total_time, response_length=0)
                # In this example, I've hardcoded response_length=0. If we would want␣
↪the response length to be
                # reported correctly in the statistics, we would probably need to␣
↪hook in at a lower level

        return wrapper


class XmlRpcUser(User):
    """
    This is the abstract User class which should be subclassed. It provides an XML-
↪RPC client
    that can be used to make XML-RPC requests that will be tracked in Locust's␣
↪statistics.
    """
    abstract = True
    def __init__(self, *args, **kwargs):
        super(XmlRpcUser, self).__init__(*args, **kwargs)
        self.client = XmlRpcClient(self.host)
        self.client._locust_environment = self.environment


class ApiUser(XmlRpcUser):
    host = "http://127.0.0.1:8877/"
    wait_time = between(0.1, 1)

    @task(10)
    def get_time(self):
        self.client.get_time()

    @task(5)
    def get_random_number(self):
        self.client.get_random_number(0, 100)
```

If you've written Locust tests before, you'll recognize the class called `ApiUser` which is a normal User class that has a couple of tasks declared. However, the `ApiUser` inherits from `XmlRpcUser` that you can see right above `ApiUser`. The `XmlRpcUser` is marked as abstract using `abstract = True` which means that Locust till not try to create simulated users from that class (only of classes that extends it). `XmlRpcUser` provides an instance of

XmlRpcClient under the `client` attribute.

The `XmlRpcClient` is a wrapper around the standard library's `xmlrpc.client.ServerProxy`. It basically just proxies the function calls, but with the important addition of firing *`locust.event.Events.request_success`* and *`locust.event.Events.request_failure`* events, which will record all calls in Locust's statistics.

Here's an implementation of an XML-RPC server that would work as a server for the code above:

```
import random
import time
from xmlrpc.server import SimpleXMLRPCServer


def get_time():
    time.sleep(random.random())
    return time.time()

def get_random_number(low, high):
    time.sleep(random.random())
    return random.randint(low, high)

server = SimpleXMLRPCServer(("localhost", 8877))
print("Listening on port 8877...")
server.register_function(get_time, "get_time")
server.register_function(get_random_number, "get_random_number")
server.serve_forever()
```

4.4 Extending Locust

Locust comes with a number of events hooks that can be used to extend Locust in different ways.

Event hooks live on the Environment instance under the *`events`* attribute. However, since the Environment instance hasn't been created when locustfiles are imported, the events object can also be accessed at the module level of the locustfile through the `locust.events` variable.

Here's an example on how to set up an event listener:

```
from locust import events

@events.request_success.add_listener
def my_success_handler(request_type, name, response_time, response_length, **kw):
    print("Successfully made a request to: %s" % name)
```

Note: It's highly recommended that you add a wildcard keyword argument in your listeners (the **kw in the code above), to prevent your code from breaking if new arguments are added in a future version.

See also:

To see all available event, please see *Event hooks*.

4.4.1 Adding Web Routes

Locust uses Flask to serve the web UI and therefore it is easy to add web end-points to the web UI. By listening to the *init* event, we can retrieve a reference to the Flask app instance and use that to set up a new route:

```
from locust import events

@events.init.add_listener
def on_locust_init(web_ui, **kw):
    @web_ui.app.route("/added_page")
    def my_added_page():
        return "Another page"
```

You should now be able to start locust and browse to http://127.0.0.1:8089/added_page

4.5 Logging

Locust uses Python's built in logging framework for handling logging.

The default logging configuration that Locust applies, writes log messages directly to stderr. `--loglevel` and `--logfile` can be used to change the verbosity and/or make the log go to a file instead.

The default logging configuration installs handlers for the `root` logger as well as the `locust.*` loggers, so using the root logger in your own test scripts will put the message into the log file if `--logfile` is used:

```
import logging
logging.info("this log message will go wherever the other locust log messages go")
```

It's also possible to control the whole logging configuration in your own test scripts by using the `--skip-log-setup` option. You will then have to configure the logging yourself.

4.5.1 Options

4.5.1.1 `--skip-log-setup`

Disable Locust's logging setup. Instead, the configuration is provided by the Locust test or Python defaults.

4.5.1.2 `--loglevel`

Choose between DEBUG/INFO/WARNING/ERROR/CRITICAL. Default is INFO. The short-hand version is `-L`.

4.5.1.3 `--logfile`

Path to log file. If not set, log will go to stdout/stderr.

4.5.2 Locust loggers

Here's a table of the loggers used within Locust (for reference when configuring logging settings manually):

Logger name	Purpose
locust	The locust namespace is used for all loggers such as `locust.main`, `locust.runners`, etc.
locust.stats_logger	This logger is used to periodically print the current stats to the console. The stats does *not* go into the log file when `--logfile` is used by default.

4.6 Using Locust as a library

It's possible to use Locust as a library, instead of running Locust using the `locust` command.

To run Locust as a library you need to create an *Environment* instance:

```
from locust.env import Environment

env = Environment(user_classes=[MyTestUser])
```

The *Environment* instance's *create_local_runner*, *create_master_runner* or *create_worker_runner* can then be used to start a *Runner* instance, which can be used to start a load test:

```
env.create_local_runner()
env.runner.start(5000, hatch_rate=20)
env.runner.greenlet.join()
```

We could also use the *Environment* instance's *create_web_ui* method to start a Web UI that can be used to view the stats, and to control the runner (e.g. start and stop load tests):

```
env.create_local_runner()
env.create_web_ui()
env.web_ui.greenlet.join()
```

4.6.1 Full example

```
import gevent
from locust import HttpUser, task, between
from locust.env import Environment
from locust.stats import stats_printer
from locust.log import setup_logging

setup_logging("INFO", None)


class User(HttpUser):
    wait_time = between(1, 3)
    host = "https://docs.locust.io"

    @task
    def my_task(self):
        self.client.get("/")

    @task
    def task_404(self):
```

(continues on next page)

(continued from previous page)

```
        self.client.get("/non-existing-path")

# setup Environment and Runner
env = Environment(user_classes=[User])
env.create_local_runner()

# start a WebUI instance
env.create_web_ui("127.0.0.1", 8089)

# start a greenlet that periodically outputs the current stats
gevent.spawn(stats_printer(env.stats))

# start the test
env.runner.start(1, hatch_rate=10)

# in 60 seconds stop the runner
gevent.spawn_later(60, lambda: env.runner.quit())

# wait for the greenlets
env.runner.greenlet.join()

# stop the web server for good measures
env.web_ui.stop()
```

CHAPTER 5

API

5.1 API

5.1.1 User class

class User(*environment*)

Represents a "user" which is to be hatched and attack the system that is to be load tested.

The behaviour of this user is defined by its tasks. Tasks can be declared either directly on the class by using the *@task decorator* on methods, or by setting the *tasks attribute*.

This class should usually be subclassed by a class that defines some kind of client. For example when load testing an HTTP system, you probably want to use the *HttpUser* class.

abstract = True

If abstract is True, the class is meant to be subclassed, and locust will not spawn users of this class during a test.

on_start()

Called when a User starts running.

on_stop()

Called when a User stops running (is killed)

tasks = []

Collection of python callables and/or TaskSet classes that the Locust user(s) will run.

If tasks is a list, the task to be performed will be picked randomly.

If tasks is a *(callable,int)* list of two-tuples, or a {callable:int} dict, the task to be performed will be picked randomly, but each task will be weighted according to its corresponding int value. So in the following case, *ThreadPage* will be fifteen times more likely to be picked than *write_post*:

```
class ForumPage(TaskSet):
    tasks = {ThreadPage:15, write_post:1}
```

wait()
Make the running user sleep for a duration defined by the User.wait_time function.

The user can also be killed gracefully while it's sleeping, so calling this method within a task makes it possible for a user to be killed mid-task even if you've set a stop_timeout. If this behavour is not desired, you should make the user wait using gevent.sleep() instead.

wait_time = None
Method that returns the time (in seconds) between the execution of locust tasks. Can be overridden for individual TaskSets.

Example:

```
from locust import User, between
class MyUser(User):
    wait_time = between(3, 25)
```

weight = 10
Probability of user class being chosen. The higher the weight, the greater the chance of it being chosen.

5.1.2 HttpUser class

class HttpUser(**args*, ***kwargs*)
Represents an HTTP "user" which is to be hatched and attack the system that is to be load tested.

The behaviour of this user is defined by its tasks. Tasks can be declared either directly on the class by using the *@task decorator* on methods, or by setting the *tasks attribute*.

This class creates a *client* attribute on instantiation which is an HTTP client with support for keeping a user session between requests.

abstract = True
If abstract is True, the class is meant to be subclassed, and users will not choose this locust during a test

client = None
Instance of HttpSession that is created upon instantiation of Locust. The client supports cookies, and therefore keeps the session between HTTP requests.

5.1.3 TaskSet class

class TaskSet(*parent*)
Class defining a set of tasks that a User will execute.

When a TaskSet starts running, it will pick a task from the *tasks* attribute, execute it, and then sleep for the number of seconds returned by its *wait_time* function. If no wait_time method has been declared on the TaskSet, it'll call the wait_time function on the User by default. It will then schedule another task for execution and so on.

TaskSets can be nested, which means that a TaskSet's *tasks* attribute can contain another TaskSet. If the nested TaskSet is scheduled to be executed, it will be instantiated and called from the currently executing TaskSet. Execution in the currently running TaskSet will then be handed over to the nested TaskSet which will continue to run until it throws an InterruptTaskSet exception, which is done when *TaskSet.interrupt()* is called. (execution will then continue in the first TaskSet).

client
Reference to the `client` attribute of the root User instance.

interrupt(*reschedule=True*)
Interrupt the TaskSet and hand over execution control back to the parent TaskSet.

If *reschedule* is True (default), the parent User will immediately re-schedule, and execute, a new task.

This method should not be called by the root TaskSet (the one that is immediately, attached to the User class's *task_set* attribute), but rather in nested TaskSet classes further down the hierarchy.

on_start()
Called when a User starts executing (enters) this TaskSet

on_stop()
Called when a User stops executing this TaskSet. E.g. when TaskSet.interrupt() is called or when the User is killed

parent = None
Will refer to the parent TaskSet, or User, class instance when the TaskSet has been instantiated. Useful for nested TaskSet classes.

schedule_task(*task_callable*, *args=None*, *kwargs=None*, *first=False*)
Add a task to the User's task execution queue.

Parameters

- **task_callable** – User task to schedule.
- **args** – Arguments that will be passed to the task callable.
- **kwargs** – Dict of keyword arguments that will be passed to the task callable.
- **first** – Optional keyword argument. If True, the task will be put first in the queue.

tasks = []
Collection of python callables and/or TaskSet classes that the User(s) will run.

If tasks is a list, the task to be performed will be picked randomly.

If tasks is a *(callable,int)* list of two-tuples, or a {callable:int} dict, the task to be performed will be picked randomly, but each task will be weighted according to its corresponding int value. So in the following case, *ThreadPage* will be fifteen times more likely to be picked than *write_post*:

```
class ForumPage(TaskSet):
    tasks = {ThreadPage:15, write_post:1}
```

user = None
Will refer to the root User class instance when the TaskSet has been instantiated

wait()
Make the running user sleep for a duration defined by the Locust.wait_time function (or TaskSet.wait_time function if it's been defined).

The user can also be killed gracefully while it's sleeping, so calling this method within a task makes it possible for a user to be killed mid-task, even if you've set a stop_timeout. If this behavour is not desired you should make the user wait using gevent.sleep() instead.

wait_time()
Method that returns the time (in seconds) between the execution of tasks.

Example:

```
from locust import TaskSet, between
class Tasks(TaskSet):
    wait_time = between(3, 25)
```

5.1.4 task decorator

task(*weight=1*)
Used as a convenience decorator to be able to declare tasks for a User or a TaskSet inline in the class. Example:

```
class ForumPage(TaskSet):
    @task(100)
    def read_thread(self):
        pass

    @task(7)
    def create_thread(self):
        pass
```

5.1.5 SequentialTaskSet class

class SequentialTaskSet(**args*, ***kwargs*)
Class defining a sequence of tasks that a User will execute.

Works like TaskSet, but task weight is ignored, and all tasks are executed in order. Tasks can either be specified by setting the *tasks* attribute to a list of tasks, or by declaring tasks as methods using the @task decorator. The order of declaration decides the order of execution.

It's possible to combine a task list in the *tasks* attribute, with some tasks declared using the @task decorator. The order of declaration is respected also in that case.

client
Reference to the `client` attribute of the root User instance.

interrupt(*reschedule=True*)
Interrupt the TaskSet and hand over execution control back to the parent TaskSet.

If *reschedule* is True (default), the parent User will immediately re-schedule, and execute, a new task.

This method should not be called by the root TaskSet (the one that is immediately, attached to the User class's *task_set* attribute), but rather in nested TaskSet classes further down the hierarchy.

on_start()
Called when a User starts executing (enters) this TaskSet

on_stop()
Called when a User stops executing this TaskSet. E.g. when TaskSet.interrupt() is called or when the User is killed

schedule_task(*task_callable*, *args=None*, *kwargs=None*, *first=False*)
Add a task to the User's task execution queue.

Parameters

- **task_callable** – User task to schedule.
- **args** – Arguments that will be passed to the task callable.
- **kwargs** – Dict of keyword arguments that will be passed to the task callable.
- **first** – Optional keyword argument. If True, the task will be put first in the queue.

wait_time()
Method that returns the time (in seconds) between the execution of tasks.

Example:

```
from locust import TaskSet, between
class Tasks(TaskSet):
    wait_time = between(3, 25)
```

5.1.6 Built in wait_time functions

between(*min_wait*, *max_wait*)
Returns a function that will return a random number between min_wait and max_wait.

Example:

```
class MyUser(User):
    # wait between 3.0 and 10.5 seconds after each task
    wait_time = between(3.0, 10.5)
```

constant(*wait_time*)
Returns a function that just returns the number specified by the wait_time argument

Example:

```
class MyUser(User):
    wait_time = constant(3)
```

constant_pacing(*wait_time*)
Returns a function that will track the run time of the tasks, and for each time it's called it will return a wait time that will try to make the total time between task execution equal to the time specified by the wait_time argument.

In the following example the task will always be executed once every second, no matter the task execution time:

```
cclass MyUser(User):
    wait_time = constant_pacing(1)
    class task_set(TaskSet):
        @task
        def my_task(self):
            time.sleep(random.random())
```

If a task execution exceeds the specified wait_time, the wait will be 0 before starting the next task.

5.1.7 HttpSession class

class HttpSession(*base_url*, *request_success*, *request_failure*, **args*, ***kwargs*)
Class for performing web requests and holding (session-) cookies between requests (in order to be able to log in and out of websites). Each request is logged so that locust can display statistics.

This is a slightly extended version of python-request's `requests.Session` class and mostly this class works exactly the same. However the methods for making requests (get, post, delete, put, head, options, patch, request) can now take a *url* argument that's only the path part of the URL, in which case the host part of the URL will be prepended with the HttpSession.base_url which is normally inherited from a User class' host property.

Each of the methods for making requests also takes two additional optional arguments which are Locust specific and doesn't exist in python-requests. These are:

Parameters

- **name** – (optional) An argument that can be specified to use as label in Locust's statistics instead of the URL path. This can be used to group different URL's that are requested into a single entry in Locust's statistics.

- **catch_response** – (optional) Boolean argument that, if set, can be used to make a request return a context manager to work as argument to a with statement. This will allow the request to be marked as a fail based on the content of the response, even if the response code is ok (2xx). The opposite also works, one can use catch_response to catch a request and then mark it as successful even if the response code was not (i.e 500 or 404).

__init__(*base_url*, *request_success*, *request_failure*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

delete(*url*, ***kwargs*)
Sends a DELETE request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

get(*url*, ***kwargs*)
Sends a GET request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

head(*url*, ***kwargs*)
Sends a HEAD request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

options(*url*, ***kwargs*)
Sends a OPTIONS request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

patch(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the `Request`.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

post (*url*, *data=None*, *json=None*, ***kwargs*)
Sends a POST request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the `Request`.
- **json** – (optional) json to send in the body of the `Request`.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

put (*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the `Request`.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

request (*method*, *url*, *name=None*, *catch_response=False*, ***kwargs*)
Constructs and sends a `requests.Request`. Returns `requests.Response` object.

Parameters

- **method** – method for the new `Request` object.
- **url** – URL for the new `Request` object.
- **name** – (optional) An argument that can be specified to use as label in Locust's statistics instead of the URL path. This can be used to group different URL's that are requested into a single entry in Locust's statistics.
- **catch_response** – (optional) Boolean argument that, if set, can be used to make a request return a context manager to work as argument to a with statement. This will allow the request to be marked as a fail based on the content of the response, even if the response code is ok (2xx). The opposite also works, one can use catch_response to catch a request and then mark it as successful even if the response code was not (i.e 500 or 404).
- **params** – (optional) Dictionary or bytes to be sent in the query string for the `Request`.
- **data** – (optional) Dictionary or bytes to send in the body of the `Request`.
- **headers** – (optional) Dictionary of HTTP Headers to send with the `Request`.
- **cookies** – (optional) Dict or CookieJar object to send with the `Request`.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How long in seconds to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) Set to True by default.

- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to `False`.
- **verify** – (optional) if `True`, the SSL cert will be verified. A CA_BUNDLE path can also be provided.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

5.1.8 Response class

This class actually resides in the python-requests library, since that's what Locust is using to make HTTP requests, but it's included in the API docs for locust since it's so central when writing locust load tests. You can also look at the `Response` class at the requests documentation.

class Response

The `Response` object, which contains a server's response to an HTTP request.

apparent_encoding

The apparent encoding, provided by the chardet library.

close()

Releases the connection back to the pool. Once this method has been called the underlying `raw` object must not be accessed again.

Note: Should not normally need to be called explicitly.

content

Content of the response, in bytes.

cookies = None

A CookieJar of Cookies the server sent back.

elapsed = None

The amount of time elapsed between sending the request and the arrival of the response (as a timedelta). This property specifically measures the time taken between sending the first byte of the request and finishing parsing the headers. It is therefore unaffected by consuming the response content or the value of the `stream` keyword argument.

encoding = None

Encoding to decode with when accessing r.text.

headers = None

Case-insensitive Dictionary of Response Headers. For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header.

history = None

A list of `Response` objects from the history of the Request. Any redirect responses will end up here. The list is sorted from the oldest to the most recent request.

is_permanent_redirect

True if this Response one of the permanent versions of redirect.

is_redirect

True if this Response is a well-formed HTTP redirect that could have been processed automatically (by `Session.resolve_redirects()`).

iter_content(*chunk_size=1*, *decode_unicode=False*)

Iterates over the response data. When stream=True is set on the request, this avoids reading the content at

once into memory for large responses. The chunk size is the number of bytes it should read into memory. This is not necessarily the length of each item returned as decoding can take place.

chunk_size must be of type int or None. A value of None will function differently depending on the value of *stream*. stream=True will read data as it arrives in whatever size the chunks are received. If stream=False, data is returned as a single chunk.

If decode_unicode is True, content will be decoded using the best available encoding based on the response.

iter_lines(*chunk_size=512*, *decode_unicode=False*, *delimiter=None*)
Iterates over the response data, one line at a time. When stream=True is set on the request, this avoids reading the content at once into memory for large responses.

Note: This method is not reentrant safe.

---

json(***kwargs*)
Returns the json-encoded content of a response, if any.

Parameters ****kwargs** – Optional arguments that `json.loads` takes.

Raises **ValueError** – If the response body does not contain valid json.

links
Returns the parsed header links of the response, if any.

next
Returns a PreparedRequest for the next request in a redirect chain, if there is one.

ok
Returns True if `status_code` is less than 400, False if not.

This attribute checks if the status code of the response is between 400 and 600 to see if there was a client error or a server error. If the status code is between 200 and 400, this will return True. This is **not** a check to see if the response code is `200 OK`.

raise_for_status()
Raises stored `HTTPError`, if one occurred.

raw = None
File-like object representation of response (for advanced usage). Use of `raw` requires that `stream=True` be set on the request. This requirement does not apply for use internally to Requests.

reason = None
Textual reason of responded HTTP Status, e.g. "Not Found" or "OK".

request = None
The `PreparedRequest` object to which this is a response.

status_code = None
Integer Code of responded HTTP Status, e.g. 404 or 200.

text
Content of the response, in unicode.

If Response.encoding is None, encoding will be guessed using `chardet`.

The encoding of the response content is determined based solely on HTTP headers, following RFC 2616 to the letter. If you can take advantage of non-HTTP knowledge to make a better guess at the encoding, you should set `r.encoding` appropriately before accessing this property.

url = None
Final URL location of Response.

5.1.9 ResponseContextManager class

class ResponseContextManager(*response*, *request_success*, *request_failure*)
A Response class that also acts as a context manager that provides the ability to manually control if an HTTP request should be marked as successful or a failure in Locust's statistics

This class is a subclass of `Response` with two additional methods: *`success`* and *`failure`*.

failure(*exc*)
Report the response as a failure.

exc can be either a python exception, or a string in which case it will be wrapped inside a CatchResponseError.

Example:

```
with self.client.get("/", catch_response=True) as response:
    if response.content == b"":
        response.failure("No data")
```

success()
Report the response as successful

Example:

```
with self.client.get("/does/not/exist", catch_response=True) as response:
    if response.status_code == 404:
        response.success()
```

5.1.10 InterruptTaskSet Exception

exception InterruptTaskSet(*reschedule=True*)
Exception that will interrupt a User when thrown inside a task

5.1.11 Environment class

class Environment(**, user_classes=[], events=None, host=None, reset_stats=False, step_load=False, stop_timeout=None, catch_exceptions=True, parsed_options=None*)

catch_exceptions = True
If True exceptions that happen within running users will be catched (and reported in UI/console). If False, exeptions will be raised.

create_local_runner()
Create a *`LocalRunner`* instance for this Environment

create_master_runner(*master_bind_host='*', master_bind_port=5557*)
Create a *`MasterRunner`* instance for this Environment

Parameters

- **master_bind_host** – Interface/host that the master should use for incoming worker connections. Defaults to "*" which means all interfaces.
- **master_bind_port** – Port that the master should listen for incoming worker connections on

create_web_ui(*host=''*, *port=8089*, *auth_credentials=None*, *tls_cert=None*, *tls_key=None*)
Creates a *WebUI* instance for this Environment and start running the web server

Parameters

- **host** – Host/interface that the web server should accept connections to. Defaults to "" which means all interfaces
- **port** – Port that the web server should listen to
- **auth_credentials** – If provided (in format "username:password") basic auth will be enabled
- **tls_cert** – An optional path (str) to a TLS cert. If this is provided the web UI will be served over HTTPS
- **tls_key** – An optional path (str) to a TLS private key. If this is provided the web UI will be served over HTTPS

create_worker_runner(*master_host*, *master_port*)
Create a *WorkerRunner* instance for this Environment

Parameters

- **master_host** – Host/IP of a running master node
- **master_port** – Port on master node to connect to

events = None
Event hooks used by Locust internally, as well as to extend Locust's functionality See *Event hooks* for available events.

host = None
Base URL of the target system

parsed_options = None
Optional reference to the parsed command line options (used to pre-populate fields in Web UI)

reset_stats = False
Determines if stats should be reset once all simulated users have been spawned

runner = None
Reference to the *Runner* instance

stats = None
Reference to RequestStats instance

step_load = False
Determines if we're running in step load mode

stop_timeout = None
If set, the runner will try to stop the runnning users gracefully and wait this many seconds before killing them hard.

user_classes = []
User classes that the runner will run

web_ui = None
Reference to the WebUI instance

5.1.12 Event hooks

Locust provide event hooks that can be used to extend Locus in various ways.

The following event hooks are available under *Environment.events*, and there's also a reference to these events under `locust.events` that can be used at the module level of locust scripts (since the Environment instance hasn't been created when the locustfile is imported).

class Events

hatch_complete
Fired when all simulated users has been spawned.

Event arguments:

Parameters **user_count** – Number of users that was hatched

alias of *EventHook*

init
Fired when Locust is started, once the Environment instance and locust runner instance have been created. This hook can be used by end-users' code to run code that requires access to the Envirionment. For example to register listeners to request_success, request_failure or other events.

Event arguments:

Parameters **environment** – Environment instance

alias of *EventHook*

init_command_line_parser
Event that can be used to add command line options to Locust

Event arguments:

Parameters **parser** – ArgumentParser instance

alias of *EventHook*

quitting
Fired when the locust process is exiting

alias of *EventHook*

report_to_master
Used when Locust is running in –worker mode. It can be used to attach data to the dicts that are regularly sent to the master. It's fired regularly when a report is to be sent to the master server.

Note that the keys "stats" and "errors" are used by Locust and shouldn't be overridden.

Event arguments:

Parameters

- **client_id** – The client id of the running locust process.
- **data** – Data dict that can be modified in order to attach data that should be sent to the master.

alias of *EventHook*

request_failure
Fired when a request fails. This event is typically used to report failed requests when writing custom clients for locust.

Event arguments:

Parameters

- **request_type** – Request type method used

- **name** – Path to the URL that was called (or override name if it was used in the call to the client)
- **response_time** – Time in milliseconds until exception was thrown
- **response_length** – Content-length of the response
- **exception** – Exception instance that was thrown

alias of *EventHook*

request_success
Fired when a request is completed successfully. This event is typically used to report requests when writing custom clients for locust.

Event arguments:

Parameters

- **request_type** – Request type method used
- **name** – Path to the URL that was called (or override name if it was used in the call to the client)
- **response_time** – Response time in milliseconds
- **response_length** – Content-length of the response

alias of *EventHook*

test_start
Fired when a new load test is started. It's not fired again if the number of users change during a test. When running locust distributed the event is only fired on the master node and not on each worker node.

alias of *EventHook*

test_stop
Fired when a load test is stopped. When running locust distributed the event is only fired on the master node and not on each worker node.

alias of *EventHook*

user_error
Fired when an exception occurs inside the execution of a User class.

Event arguments:

Parameters

- **user_instance** – User class instance where the exception occurred
- **exception** – Exception that was thrown
- **tb** – Traceback object (from sys.exc_info()[2])

alias of *EventHook*

worker_report
Used when Locust is running in –master mode and is fired when the master server receives a report from a Locust worker server.

This event can be used to aggregate data from the locust worker servers.

Event arguments:

Parameters

- **client_id** – Client id of the reporting worker

- **data** – Data dict with the data from the worker node

alias of *EventHook*

5.1.12.1 EventHook class

The event hooks are instances of the **locust.events.EventHook** class:

class EventHook

Simple event class used to provide hooks for different types of events in Locust.

Here's how to use the EventHook class:

```
my_event = EventHook()
def on_my_event(a, b, **kw):
    print("Event was fired with arguments: %s, %s" % (a, b))
my_event.add_listener(on_my_event)
my_event.fire(a="foo", b="bar")
```

If reverse is True, then the handlers will run in the reverse order that they were inserted

---

Note: It's highly recommended that you add a wildcard keyword argument in your event listeners to prevent your code from breaking if new arguments are added in a future version.

5.1.13 Runner classes

class Runner(*environment*)

Orchestrates the load test by starting and stopping the users.

Use one of the *create_local_runner*, *create_master_runner* or *create_worker_runner* methods on the *Environment* instance to create a runner of the desired type.

quit()

Stop any running load test and kill all greenlets for the runner

start(*user_count*, *hatch_rate*, *wait=False*)

Start running a load test

Parameters

- **user_count** – Number of users to start
- **hatch_rate** – Number of users to spawn per second
- **wait** – If True calls to this method will block until all users are spawned. If False (the default), a greenlet that spawns the users will be started and the call to this method will return immediately.

stop()

Stop a running load test by stopping all running users

user_count

Returns Number of currently running users

class LocalRunner(*environment*)

Runner for running single process load test

class MasterRunner(*environment*, *master_bind_host*, *master_bind_port*)
Runner used to run distributed load tests across multiple processes and/or machines.

MasterRunner doesn't spawn any user greenlets itself. Instead it expects *WorkerRunners* to connect to it, which it will then direct to start and stop user greenlets. Stats sent back from the *WorkerRunners* will aggregated.

class WorkerRunner(*environment*, *master_host*, *master_port*)
Runner used to run distributed load tests across multiple processes and/or machines.

WorkerRunner connects to a *MasterRunner* from which it'll receive instructions to start and stop user greenlets. The WorkerRunner will preiodically take the stats generated by the running users and send back to the *MasterRunner*.

5.1.14 Web UI class

class WebUI(*environment*, *host*, *port*, *auth_credentials=None*, *tls_cert=None*, *tls_key=None*)
Sets up and runs a Flask web app that can start and stop load tests using the *environment.runner* as well as show the load test statistics in *environment.stats*

app = None
Reference to the `flask.Flask` app. Can be used to add additional web routes and customize the Flask app in other various ways. Example:

```
from flask import request

@web_ui.app.route("/my_custom_route")
def my_custom_route():
    return "your IP is: %s" % request.remote_addr
```

auth_required_if_enabled(*view_func*)
Decorator that can be used on custom route methods that will turn on Basic Auth authentication if the `--web-auth` flag is used. Example:

```
@web_ui.app.route("/my_custom_route")
@web_ui.auth_required_if_enabled
def my_custom_route():
    return "custom response"
```

greenlet = None
Greenlet of the running web server

server = None
Reference to the `pyqsgi.WSGIServer` instance

stop()
Stop the running web server

CHAPTER 6

Third-party tools

6.1 Third party tools

6.1.1 Support for other sampler protocols, reporting etc

- Locust Plugins

6.1.2 Automate distributed runs with no manual steps

- Locust Swarm

6.1.3 Using other languages

A Locust master and a Locust worker communicate by exchanging msgpack messages, which is supported by many languages. So, you can write your User tasks in any languages you like. For convenience, some libraries do the job as a worker runner. They run your User tasks, and report to master regularly.

6.1.3.1 Golang

- Boomer

6.1.3.2 Java

- Locust4j
- Swarm

6.1.4 Configuration Management

Deploying Locust is easy, but here and there some tools can still provide a measure of convenience.

tinx.locust is an Ansible role to install, configure and control Locust as a systemd service, or to build Locust docker images using ansible-container. Also manages locustfiles and accompanying test data.

CHAPTER 7

Changelog

7.1 Changelog Highlights

For full details of the Locust changelog, please see https://github.com/locustio/locust/blob/master/CHANGELOG.md

7.1.1 1.0 (in development, 1.0b1 released)

This version contains some breaking changes.

7.1.1.1 Locust class renamed to User

We've renamed the `Locust` and `HttpLocust` classes to `User` and `HttpUser`. The `locust` attribute on *`TaskSet`* instances has been renamed to *`user`*.

7.1.1.2 Ability to declare @task directly under the `User` class

It's now possible to declare tasks directly under a User class like this:

```
class WebUser(User):
    @task
    def some_task(self):
        pass
```

In tasks declared under a User class (e.g. `some_task` in the example above), `self` refers to the User instance, as one would expect. For tasks defined under a *`TaskSet`* class, `self` would refer to the `TaskSet` instance.

The `task_set` attribute on the `User` class (previously `Locust` class) has been removed. To declare a `User` class with a single `TaskSet` one would now use the the *`tasks`* attribute instead:

```
class MyTaskSet(TaskSet):
    ...

class WebUser(User):
    tasks = [MyTaskSet]
```

7.1.1.3 Environment variables changed

The following changes has been made to the configuration environment variables

- `LOCUST_MASTER` has been renamed to `LOCUST_MODE_MASTER` (in order to make it less likely to get variable name collisions when running Locust in Kubernetes/K8s which automatically adds environment variables depending on service/pod names).
- `LOCUST_SLAVE` has been renamed to `LOCUST_MODE_WORKER`.
- `LOCUST_MASTER_PORT` has been renamed to `LOCUST_MASTER_NODE_PORT`.
- `LOCUST_MASTER_HOST` has been renamed to `LOCUST_MASTER_NODE_HOST`.
- `CSVFILEBASE` has been renamed to `LOCUST_CSV`.

See the *Configuration* documentation for a full list of available *environment variables*.

7.1.1.4 Other breaking changes

- The master/slave terminology has been changed to master/worker. Therefore the command line arguments `--slave` and `--expect-slaves` has been renamed to `--worker` and `--expect-workers`.
- The option for running Locust without the Web UI has been renamed from `--no-web` to `--headless`.
- Removed `Locust.setup`, `Locust.teardown`, `TaskSet.setup` and `TaskSet.teardown` hooks. If you want to run code at the start or end of a test, you should instead use the *`test_start`* and *`test_stop`* events:

```
from locust import events

@events.test_start.add_listener
def on_test_start(**kw):
    print("test is starting")

@events.test_stop.add_listener
def on_test_start(**kw):
    print("test is stopping")
```

- `TaskSequence` and `@seq_task` has been replaced with *SequentialTaskSet*.
- A `User count` column has been added to the history stats CSV file. The column order and column names has been changed.
- The official docker image no longer uses a shell script with a bunch of special environment variables to configure how how locust is started. Instead, the `locust` command is now set as `ENTRYPOINT` of the docker image. See *Running Locust with Docker* for more info.
- Command line option `--csv-base-name` has been removed, since it was just an alias for `--csv`.
- The way Locust handles logging has been changed. We no longer wrap stdout (and stderr) to automatically make print statements go into the log. `print()` statements now only goes to stdout. To add custom entries to the log, one should now use the Python logging module:

```
import logging
logging.info("custom logging message)
```

For more info see *Logging*

7.1.1.5 Web UI improvements

- It's now possible to protect the Web UI with Basic Auth using hte `--web-auth` command line argument.
- The Web UI can now be served over HTTPS by specifying a TLS certificate and key with the `--tls-cert` and `--tls-key` command line arguments.
- If the number of users and hatch rate are specified on command line, it's now used to pre-populate the input fields in the Web UI.

7.1.1.6 Other fixes and improvements

- Added `--config` command line option for specifying a *configuration file* path
- The code base has been refactored to make it possible to run *Locust as a python lib*.
- It's now possible to call `response.failure()` or `response.success()` multiple times when using the `catch_response=True` in the HTTP clients. Only the last call to `success`/`failure` will count.
- The `--help` output has been improved by grouping related options together.

7.1.2 0.14.6

- Fix bug when running with latest Gevent version, and pinned the latest version

7.1.3 0.14.0

- Drop Python 2 and Python 3.5 support!
- Continuously measure CPU usage and emit a warning if we get a five second average above 90%
- Show CPU usage of slave nodes in the Web UI
- Fixed issue when running Locust distributed and new slave nodes connected during the hatching/ramp-up phase (https://github.com/locustio/locust/issues/1168)

7.1.4 0.13.5

Various minor fixes, mainly regarding FastHttpLocust.

7.1.5 0.13.4

Identical to previous version, but now built & deployed to Pypi using Travis.

7.1.6 0.13.3

- Unable to properly connect multiple slaves - https://github.com/locustio/locust/issues/1176
- Zero exit code on exception - https://github.com/locustio/locust/issues/1172
- *–stop-timeout* is not respected when changing number of running Users in distributed mode - https://github.com/locustio/locust/issues/1162

7.1.7 0.13.2

- Fixed bug that broke the Web UI's repsonse time graph

7.1.8 0.13.1

- Fixed crash bug on Python 3.8.0
- Various other bug fixes and improvements.

7.1.9 0.13.0

- New API for specifying wait time - https://github.com/locustio/locust/pull/1118

  Example of the new API:

```
from locust import HttpLocust, between
class User(HttpLocust):
    # wait between 5 and 30 seconds
    wait_time = between(5, 30)
```

  There are three built in *wait time functions*: `between`, `constant` and `constant_pacing`.
- FastHttpLocust: Accept self signed SSL certificates, ignore host checks. Improved response code handling
- Add current working dir to sys.path - https://github.com/locustio/locust/pull/484
- Web UI improvements: Added 90th percentile to table, failure per seconds as a series in the chart
- Ability to specify host in web ui
- Added response_length to request_failure event - https://github.com/locustio/locust/pull/1144
- Added p99.9 and p99.99 to request stats distribution csv - https://github.com/locustio/locust/pull/1125
- Various other bug fixes and improvements.

7.1.10 0.12.2

- Added *–skip-log-setup* to disable Locust's default logging setup.
- Added *–stop-timeout* to allow tasks to finish running their iteration before stopping
- Added 99.9 and 99.99 percentile response times to csv output
- Allow custom clients to set request response time to None. Those requests will be excluded when calculating median, average, min, max and percentile response times.
- Renamed the last row in statistics table from "Total" to "Aggregated" (since the values aren't a sum of the individual table rows).

- Some visual improvements to the web UI.
- Fixed issue with simulating fewer number of locust users than the number of slave/worker nodes.
- Fixed bugs in the web UI related to the fact that the stats table is truncated at 500 entries.
- Various other bug fixes and improvements.

7.1.11 0.12.1

- Added new `FastHttpLocust` class that uses a faster HTTP client, which should be 5-6 times faster than the normal `HttpLocust` class. For more info see the documentation on *increasing performance*.
- Added ability to set the exit code of the locust process when exceptions has occurred within the user code, using the `--exit-code-on-error` parameter.
- Added TCP keep alive to master/slave communication sockets to avoid broken connections in some environments.
- Dropped support for Python 3.4
- Numerous other bug fixes and improvements.

7.1.12 0.10.0

- Python 3.7 support
- Added a status page to the web UI when running Locust distributed showing the status of slave nodes and detect down slaves using heartbeats
- Numerous bugfixes/documentation updates (see detailed changelog)

7.1.13 0.9.0

- Added detailed changelog (https://github.com/locustio/locust/blob/master/CHANGELOG.md)
- Numerous bugfixes (see detailed changelog)
- Added sequential task support - https://github.com/locustio/locust/pull/827
- Added support for user-defined wait_function - https://github.com/locustio/locust/pull/785
- By default, Locust no longer resets the statistics when the hatching is complete. Therefore `--no-reset-stats` has been deprected (since it's now the default behaviour), and instead a new `--reset-stats` option has been added.
- Dropped support for Python 3.3
- Updated documentation

7.1.14 0.8.1

- Updated pyzmq version, and changed so that we don't pin a specific version. This makes it easier to install Locust on Windows.

7.1.15 0.8

- Python 3 support
- Dropped support for Python 2.6
- Added `--no-reset-stats` option for controling if the statistics should be reset once the hatching is complete
- Added charts to the web UI for requests per second, average response time, and number of simulated users.
- Updated the design of the web UI.
- Added ability to write a CSV file for results via command line flag
- Added the URL of the host that is currently being tested to the web UI.
- We now also apply gevent's monkey patching of threads. This fixes an issue when using Locust to test Cassandra (https://github.com/locustio/locust/issues/569).
- Various bug fixes and improvements

7.1.16 0.7.5

- Use version 1.1.1 of gevent. Fixes an install issue on certain versions of python.

7.1.17 0.7.4

- Use a newer version of requests, which fixed an issue for users with older versions of requests getting ConnectionErrors (https://github.com/locustio/locust/issues/273).
- Various fixes to documentation.

7.1.18 0.7.3

- Fixed bug where POST requests (and other methods as well) got incorrectly reported as GET requests, if the request resulted in a redirect.
- Added ability to download exceptions in CSV format. Download links has also been moved to it's own tab in the web UI.

7.1.19 0.7.2

- Locust now returns an exit code of 1 when any failed requests were reported.
- When making an HTTP request to an endpoint that responds with a redirect, the original URL that was requested is now used as the name for that entry in the statistics (unless an explicit override is specified through the *name* argument). Previously, the last URL in the redirect chain was used to label the request(s) in the statistics.
- Fixed bug which caused only the time of the last request in a redirect chain to be included in the reported time.
- Fixed bug which caused the download time of the request body not to be included in the reported response time.
- Fixed bug that occurred on some linux dists that were tampering with the python-requests system package (removing dependencies which requests is bundling). This bug only occured when installing Locust in the python system packages, and not when using virtualenv.
- Various minor fixes and improvements.

7.1.20 0.7.1

- Exceptions that occurs within TaskSets are now catched by default.
- Fixed bug which caused Min response time to always be 0 after all locusts had been hatched and the statistics had been reset.
- Minor UI improvements in the web interface.
- Handle messages from "zombie" slaves by ignoring the message and making a log entry in the master process.

7.1.21 0.7

7.1.21.1 HTTP client functionality moved to HttpLocust

Previously, the Locust class instantiated a *HttpSession* under the client attribute that was used to make HTTP requests. This funcionality has now been moved into the HttpLocust class, in an effort to make it more obvious how one can use Locust to *load test non-HTTP systems*.

To make existing locust scripts compatible with the new version you should make your locust classes inherit from HttpLocust instead of the base Locust class.

7.1.21.2 msgpack for serializing master/slave data

Locust now uses msgpack for serializing data that is sent between a master node and it's slaves. This adresses a possible attack that can be used to execute code remote, if one has access to the internal locust ports that are used for master-slave communication. The reason for this exploit was due to the fact that pickle was used.

Warning: Anyone who uses an older version should make sure that their Locust machines are not publicly accessible on port 5557 and 5558. Also, one should never run Locust as root.

Anyone who uses the report_to_master and slave_report events, needs to make sure that any data that is attached to the slave reports is serializable by msgpack.

7.1.21.3 requests updated to version 2.2

Locust updated requests to the latest major release.

Note: Requests 1.0 introduced some major API changes (and 2.0 just a few). Please check if you are using any internal features and check the documentation: Migrating to 1.x and Migrationg to 2.x

7.1.21.4 gevent updated to version 1.0

gevent 1.0 has now been released and Locust has been updated accordingly.

7.1.21.5 Big refactoring of request statistics code

Refactored `RequestStats`.

- Created `StatsEntry` which represents a single stats entry (URL).

Previously the `RequestStats` was actually doing two different things:

- It was holding track of the aggregated stats from all requests
- It was holding the stats for single stats entries.

Now RequestStats should be instantiated and holds the global stats, as well as a dict of StatsEntry instances which holds the stats for single stats entries (URLs)

7.1.21.6 Removed support for avg_wait

Previously one could specify avg_wait to `TaskSet` and `Locust` that Locust would try to strive to. However this can be sufficiently accomplished by using min_wait and max_wait for most use-cases. Therefore we've decided to remove the avg_wait as it's use-case is not clear or just too narrow to be in the Locust core.

7.1.21.7 Removed support for ramping

Previously one could tell Locust, using the –ramp option, to try to find a stable client count that the target host could handle, but it's been broken and undocumented for quite a while so we've decided to remove it from the locust core and perhaps have it reappear as a plugin in the future.

7.1.21.8 Locust Event hooks now takes keyword argument

When *Extending Locust* by listening to *Event hooks*, the listener functions should now expect the arguments to be passed in as keyword arguments. It's also highly recommended to add an extra wildcard keyword arguments to listener functions, since they're then less likely to break if extra arguments are added to that event in some future version. For example:

```
from locust import events

def on_request(request_type, name, response_time, response_length, **kw):
    print "Got request!"

locust.events.request_success += on_request
```

The *method* and *path* arguments to `request_success` and `request_failure` are now called *request_type* and *name*, since it's less HTTP specific.

7.1.21.9 Other changes

- You can now specify the port on which to run the web host
- Various code cleanups
- Updated gevent/zmq libraries
- Switched to unittest2 discovery
- Added option –only-summary to only output the summary to the console, thus disabling the periodic stats output.

- Locust will now make sure to spawn all the specified locusts in distributed mode, not just a multiple of the number of slaves.
- Fixed the broken Vagrant example.
- Fixed the broken events example (events.py).
- Fixed issue where the request column was not sortable in the web-ui.
- Minor styling of the statistics table in the web-ui.
- Added options to specify host and ports in distributed mode using –master-host, –master-port for the slaves, –master-bind-host, –master-bind-port for the master.
- Removed previously deprecated and obsolete classes WebLocust and SubLocust.
- Fixed so that also failed requests count, when specifying a maximum number of requests on the command line

7.1.22 0.6.2

- Made Locust compatible with gevent 1.0rc2. This allows user to step around a problem with running Locust under some versions of CentOS, that can be fixed by upgrading gevent to 1.0.
- Added `parent` attribute to TaskSet class that refers to the parent TaskSet, or Locust, instance. Contributed by Aaron Daubman.

7.1.23 0.6.1

- Fixed bug that was causing problems when setting a maximum number of requests using the **-n** or **–num-request** command line parameter.

7.1.24 0.6

Warning: This version comes with non backward compatible changes to the API. Anyone who is currently using existing locust scripts and want to upgrade to 0.6 should read through these changes.

7.1.24.1 `SubLocust` replaced by `TaskSet` and `Locust` class behaviour changed

`Locust` classes does no longer control task scheduling and execution. Therefore, you no longer define tasks within Locust classes, instead the Locust class has a `task_set` attribute which should point to a `TaskSet` class. Tasks should now be defined in TaskSet classes, in the same way that was previously done in Locust and SubLocust classes. TaskSets can be nested just like SubLocust classes could.

So the following code for 0.5.1:

```
class User(Locust):
    min_wait = 10000
    max_wait = 120000

    @task(10)
    def index(self):
        self.client.get("/")

    @task(2)
```

(continues on next page)

(continued from previous page)

```
    class AboutPage(SubLocust):
        min_wait = 10000
        max_wait = 120000

        def on_init(self):
            self.client.get("/about/")

        @task
        def team_page(self):
            self.client.get("/about/team/")

        @task
        def press_page(self):
            self.client.get("/about/press/")

        @task
        def stop(self):
            self.interrupt()
```

Should now be written like:

```
class BrowsePage(TaskSet):
    @task(10)
    def index(self):
        self.client.get("/")

    @task(2)
    class AboutPage(TaskSet):
        def on_init(self):
            self.client.get("/about/")

        @task
        def team_page(self):
            self.client.get("/about/team/")

        @task
        def press_page(self):
            self.client.get("/about/press/")

        @task
        def stop(self):
            self.interrupt()

class User(Locust):
    min_wait = 10000
    max_wait = 120000
    task_set = BrowsePage
```

Each TaskSet instance gets a `locust` attribute, which refers to the Locust class.

7.1.24.2 Locust now uses Requests

Locust's own HttpBrowser class (which was typically accessed through *self.client* from within a locust class) has been replaced by a thin wrapper around the requests library (http://python-requests.org). This comes with a number of advantages. Users can now take advantage of a well documented, well written, fully fledged library for making HTTP

requests. However, it also comes with some small API changes wich will require users to update their existing load testing scripts.

Gzip encoding turned on by default

The HTTP client now sends headers for accepting gzip encoding by default. The **–gzip** command line argument has been removed and if someone want to disable the *Accept-Encoding* that the HTTP client uses, or any other HTTP headers you can do:

```
class MyWebUser(Locust):
    def on_start(self):
        self.client.headers = {"Accept-Encoding":""}
```

Improved HTTP client

Because of the switch to using python-requests in the HTTP client, the API for the client has also gotten a few changes.

- Additionally to the *get*, *post*, *put*, *delete* and *head* methods, the *HttpSession* class now also has *patch* and *options* methods.
- All arguments to the HTTP request methods, except for **url** and **data** should now be specified as keyword arguments. For example, previously one could specify headers using:

  ```
  client.get("/path", {"User-Agent":"locust"}) # this will no longer work
  ```

  And should now be specified like:

  ```
  client.get("/path", headers={"User-Agent":"locust"})
  ```

- In general the whole HTTP client is now more powerful since it leverages on python-requests. Features that we're now able to use in Locust includes file upload, SSL, connection keep-alive, and more. See the python-requests documentation for more details.
- The new *HttpSession* class' methods now return python-request `Response` objects. This means that accessing the content of the response is no longer made using the **data** attribute, but instead the **content** attribute. The HTTP response code is now accessed through the **status_code** attribute, instead of the **code** attribute.

HttpSession methods' catch_response argument improved and allow_http_error argument removed

- When doing HTTP requests using the **catch_response** argument, the context manager that is returned now provides two functions, *success* and *failure* that can be used to manually control what the request should be reported as in Locust's statistics.

  class ResponseContextManager(*response*, *request_success*, *request_failure*)
  A Response class that also acts as a context manager that provides the ability to manually control if an HTTP request should be marked as successful or a failure in Locust's statistics

  This class is a subclass of `Response` with two additional methods: *success* and *failure*.

  failure(*exc*)
  Report the response as a failure.

  exc can be either a python exception, or a string in which case it will be wrapped inside a CatchResponseError.

  Example:

```
with self.client.get("/", catch_response=True) as response:
    if response.content == b"":
        response.failure("No data")
```

success()
Report the response as successful

Example:

```
with self.client.get("/does/not/exist", catch_response=True) as response:
    if response.status_code == 404:
        response.success()
```

- The **allow_http_error** argument of the HTTP client's methods has been removed. Instead one can use the **catch_response** argument to get a context manager, which can be used together with a with statement.

  The following code in the previous Locust version:

```
client.get("/does/not/exist", allow_http_error=True)
```

  Can instead now be written like:

```
with client.get("/does/not/exist", catch_response=True) as response:
    response.success()
```

7.1.24.3 Other improvements and bug fixes

- Scheduled task callables can now take keyword arguments and not only normal function arguments.
- SubLocust classes that are scheduled using `locust.core.Locust.schedule_task()` can now take arguments and keyword arguments (available in *self.args* and *self.kwargs*).
- Fixed bug where the average content size would be zero when doing requests against a server that didn't set the content-length header (i.e. server that uses *Transfer-Encoding: chunked*)

7.1.24.4 Smaller API Changes

- The *require_once* decorator has been removed. It was an old legacy function that no longer fit into the current way of writing Locust tests, where tasks are either methods under a Locust class or SubLocust classes containing task methods.
- Changed signature of `locust.core.Locust.schedule_task()`. Previously all extra arguments that was given to the method was passed on to the task when it was called. It no longer accepts extra arguments. Instead, it takes an *args* argument (list) and a *kwargs* argument (dict) which are be passed to the task when it's called.
- Arguments for `request_success` event hook has been changed. Previously it took an HTTP Response instance as argument, but this has been changed to take the content-length of the response instead. This makes it easier to write custom clients for Locust.

7.1.25 0.5.1

- Fixed bug which caused –logfile and –loglevel command line parameters to not be respected when running locust without zeromq.

7.1.26 0.5

7.1.26.1 API changes

- Web inteface is now turned on by default. The **–web** command line option has been replaced by –no-web.
- `locust.events.request_success()` and `locust.events.request_failure()` now gets the HTTP method as the first argument.

7.1.26.2 Improvements and bug fixes

- Removed **–show-task-ratio-confluence** and added a **–show-task-ratio-json** option instead. The **–show-task-ratio-json** will output JSON data containing the task execution ratio for the locust "brain".
- The HTTP method used when a client requests a URL is now displayed in the web UI
- Some fixes and improvements in the stats exporting:
- A file name is now set (using content-disposition header) when downloading stats.
- The order of the column headers for request stats was wrong.
- Thanks Benjamin W. Smith, Jussi Kuosa and Samuele Pedroni!

7.1.27 0.4

7.1.27.1 API changes

- WebLocust class has been deprecated and is now called just Locust. The class that was previously called Locust is now called LocustBase.
- The *catch_http_error* argument to HttpClient.get() and HttpClient.post() has been renamed to *allow_http_error*.

7.1.27.2 Improvements and bug fixes

- Locust now uses python's logging module for all logging
- Added the ability to change the number of spawned users when a test is running, without having to restart the test.
- Experimental support for automatically ramping up and down the number of locust to find a maximum number of concurrent users (based on some parameters like response times and acceptable failure rate).
- Added support for failing requests based on the response data, even if the HTTP response was OK.
- Improved master node performance in order to not get bottlenecked when using enough slaves (>100)
- Minor improvements in web interface.
- Fixed missing template dir in MANIFEST file causing locust installed with "setup.py install" not to work.

Python Module Index

l

Index