pytest-{{cookiecutter.plugin_name}} Documentation

Release 2.3.0

{{cookiecutter.full_name}}

Jun 12, 2019

Contents

`pytest-play` is a codeless, generic, pluggable and extensible **automation tool**, not necessarily **test automation** only, based on the fantastic pytest test framework that let you define and execute YAML files containing scripts or test scenarios through actions and assertions that can be implemented and managed even by **non technical users**:

- automation (not necessarily test automation). You can build a set of actions on a single file (e.g, call a JSON based API endpoint, perform an action if a condition matches) or a test automation project with many test scenarios.

  For example you can create always fresh test data on demand supporting manual testing activities, build a live simulator and so on

- codeless, or better almost codeless. If you have to write assertions against action results or some conditional expressions you need a very basic knowledge of Python or Javascript expressions with a smooth learning curve (something like `variables['foo'] == 'bar'`)

- generic. It is not yet again another automation tool for browser automation only, API only, etc. You can drive a browser, perform some API calls, make database queries and/or make assertions using the same tool for different technologies

  So there are several free or not free testing frameworks or automation tools and many times they address just one single area testing needs and they are not extensible: API testing only, UI testing only and so on. It could be fine if you are testing a web only application like a CMS but if you are dealing with a reactive IoT application you might something more, make cross actions or cross checks against different systems or build something of more complex upon `pytest-play`

- powerful. It is not yet again another test automation tool, it only extends the pytest framework with another paradigm and inherits a lot of good stuff (test data decoupled by test implementation that let you write once and executed many times the same scenario thanks to native parametrization support, reporting, integration with test management tools, many useful command line options, browsers and remote Selenium grids integration, etc)

- pluggable and extensible. Let's say you need to interact with a system not yet supported by a `pytest-play` plugin, you can write by your own or pay someone for you. In addition there is a scaffolding tool that let you implement your own command: https://github.com/davidemoro/cookiecutter-play-plugin

- easy to use. Why YAML? Easy to read, easy to write, simple and standard syntax, easy to be validated and no parentheses hell. Despite there are no recording tools (not yet) for browser interaction or API calls, the documentation based on very common patterns let you copy, paste and edit command by command with no pain

- free software. It's an open source project based on the large and friendly pytest community

- easy to install. The only prerequisite is Docker thanks to the `davidemoro/pytest-play` Docker Hub container. Or better, with docker, no installation is required: you just need to type the following command `docker run -i --rm -v $(pwd):/src davidemoro/pytest-play` inside your project folder See https://hub.docker.com/r/davidemoro/pytest-play

See at the bottom of the page the third party plugins that extends `pytest-play`:

- *Third party pytest-play plugins*

CHAPTER 1

How it works

Depending on your needs and skills you can choose to use pytest-play programmatically writing some Python code or following a Python-less approach.

As said before with *pytest-play* you will be able to create codeless scripts or test scenarios with no or very little Python knowledge: a file `test_XXX.yml` (e.g., `test_something.yml`, where `test_` and `.yml` matter) will be automatically recognized and executed without having to touch any `*.py` module.

You can run a single scenario with `pytest test_XXX.yml` or running the entire suite filtering by name or keyword markers.

Despite `pytest-play` was born with native support for JSON format, `pytest-play`>=2.0 versions will support YAML only for improved usability.

CHAPTER 2

Python-less (pure YAML)

Here you can see the contents of a `pytest-play` project without any Python files inside containing a login scenario:

```
$ tree
.
├── env-ALPHA.yml    (OPTIONAL)
└── test_login.yml
```

and you might have some global variables in a settings file specific for a target environment:

```
$ cat env-ALPHA.yml
pytest-play:
  base_url: https://www.yoursite.com
```

The test scenario with action, assertions and optional metadata (play_selenium external plugin needed):

```
$ cat test_login.yml
---
markers:
  - login
test_data:
  - username: siteadmin
    password: siteadmin
  - username: editor
    password: editor
  - username: reader
    password: reader
---
- comment: visit base url
  type: get
  provider: selenium
  url: "$base_url"
- comment: click on login link
  locator:
    type: id
```

(continues on next page)

(continued from previous page)

```
    value: personaltools-login
  type: clickElement
  provider: selenium
- comment: provide a username
  locator:
    type: id
    value: __ac_name
  text: "$username"
  type: setElementText
  provider: selenium
- comment: provide a password
  locator:
    type: id
    value: __ac_password
  text: "$password"
  type: setElementText
  provider: selenium
- comment: click on login submit button
  locator:
    type: css
    value: ".pattern-modal-buttons > input[name=submit]"
  type: clickElement
  provider: selenium
- comment: wait for page loaded
  locator:
    type: css
    value: ".icon-user"
  type: waitForElementVisible
  provider: selenium
```

The first optional YAML document contains some metadata with keywords aka `markers` so you can filter tests to be executed invoking pytest with marker expressions, decoupled test data, etc.

The same `test_login.yml` scenario will be executed 3 times with different decoupled test data `test_data` defined inside its first optional YAML document (the block between the 2 `---` lines).

So write once and execute many times with different test data!

You can see a hello world example here:

- https://github.com/davidemoro/pytest-play-plone-example

As told before the metadata document is optional so you might have 1 or 2 documents in your YAML file. You can find more info about *Metadata format*.

Here you can see the same example without the metadata section for sake of completeness:

```
---
- comment: visit base url
  type: get
  provider: selenium
  url: "http://YOURSITE"
- comment: click on login link
  type: clickElement
  provider: selenium
  locator:
    type: id
    value: personaltools-login
```

(continues on next page)

(continued from previous page)

```
- comment: provide a username
  type: setElementText
  provider: selenium
  locator:
    type: id
    value: __ac_name
  text: "YOURUSERNAME"
- comment: provide a password
  type: setElementText
  provider: selenium
  locator:
    type: id
    value: __ac_password
  text: "YOURPASSWORD"
- comment: click on login submit button
  type: clickElement
  provider: selenium
  locator:
    type: css
    value: ".pattern-modal-buttons > input[name=submit]"
- comment: wait for page loaded
  type: waitForElementVisible
  provider: selenium
  locator:
    type: css
    value: ".icon-user"
```

CHAPTER 3

Programmatically

You can invoke pytest-play programmatically too.

You can define a test `test_login.py` like this:

```
def test_login(play):
    data = play.get_file_contents(
        'my', 'path', 'etc', 'login.yml')
    play.execute_raw(data, extra_variables={})
```

Or this programmatical approach might be used if you are implementing BDD based tests using `pytest-bdd`.

3.1 Core commands

pytest-play provides some core commands that let you:

- write simple Python assertions, expressions and variables
- reuse steps including other test scenario scripts
- provide a default command template for some particular providers (eg: add by default HTTP authentication headers for all requests)
- a generic wait until machinery. Useful for waiting for an observable asynchronous event will complete its flow before proceeding with the following commands that depends on the previous step completion

You can write restricted Python expressions and assertions based on the `RestrictedPython` package.

RestrictedPython is a tool that helps to define a subset of the Python language which allows to provide a program input into a trusted environment. RestrictedPython is not a sandbox system or a secured environment, but it helps to define a trusted environment and execute untrusted code inside of it.

See:

- https://github.com/zopefoundation/RestrictedPython

CHAPTER 4

How to reuse steps

You can split your commands and reuse them using the `include` command avoiding duplication:

```
- provider: include
  type: include
  path: "/some-path/included-scenario.yml"
```

You can create a variable for the base folder where your test scripts live.

CHAPTER 5

Default commands

Some commands require many verbose options you don't want to repeat (eg: authentication headers for play_requests).

Instead of replicating all the headers information you can initialize a `pytest-play` with the provider name as key and as a value the default command you want to omit (this example neets the external plugin play_selenium):

```
- provider: python
  type: store_variable
  name: bearer
  expression: "'BEARER'"
- provider: python
  type: store_variable
  name: play_requests
  expression: "{'parameters': {'headers': {'Authorization': '$bearer'}}}"
- provider: play_requests
  type: GET
  comment: this is an authenticated request!
  url: "$base_url"
```

CHAPTER 6

Store variables

You can store a *pytest-play* variables:

```
- provider: python
  type: store_variable
  expression: "1+1"
  name: foo
```

CHAPTER 7

Make a Python assertion

You can make an assertion based on a Python expression:

```
- provider: python
  type: assert
  expression: variables['foo'] == 2
```

CHAPTER 8

Sleep

Sleep for a given amount of seconds:

```
- provider: python
  type: sleep
  seconds: 2
```

CHAPTER 9

Exec a Python expresssion

You can execute a Python expression:

```
- provider: python
  type: exec
  expression: "1+1"
```

CHAPTER 10

While condition and looping

If you need to loop over a series of commands or wait something you can use the `while` command. It will execute the sequence of sub commands, if any, while the resulting expression condition is true. Assuming you have a `countdown` variable containing a integer `10`, the block of commands whill be executed 10 times:

```
---
- provider: python
  type: while
  expression: variables['countdown'] >= 0
  timeout: 2.3
  poll: 0.1
  sub_commands:
  - provider: python
    type: store_variable
    name: countdown
    expression: variables['countdown'] - 1
```

The `while` command supersedes the other legacy commands `wait_until` (stops when the condition becomes true) or `wait_until_not`. commands.

CHAPTER 11

Conditional commands (Python)

You can skip any command evaluating a Python based skip condition like the following:

```
- provider: include
  type: include
  path: "/some-path/assertions.yml"
  skip_condition: variables['cassandra_assertions'] is True
```

CHAPTER 12

How to assert commands elapsed time

The engine updates a `pytest-play` variable called `_elapsed` for each executed command. So you can write something that:

```
---
- type: GET
  provider: play_requests
  url: https://api.chucknorris.io/jokes/categories
  expression: "'dev' in response.json()"
- type: assert
  provider: python
  expression: "variables['_elapsed'] > 0"
```

CHAPTER 13

Generate a JUnit XML report

Use the `--junit-xml` command line option, e.g.:

```
--junit-xml results.xml
```

You'll get for each test case errors, commands executed in `system-output` (do not use `-s` or `--capture=no` otherwise you won't see commands in `system-output`) and execution timing metrics (global, per test case and per single command thanks to `_elapsed` property tracked on every executed command shown in `system-output`).

Here you can see a standard `results.xml` file:

```
<?xml version="1.0" encoding="utf-8"?><testsuite errors="0" failures="0" name="pytest
↪" skipped="0" tests="1" time="0.360"><testcase classname="test_assertion.yml" file=
↪"test_assertion.yml" name="test_assertion.yml" time="0.326"><system-out>{&apos;
↪expression&apos;: &apos;1 == 1&apos;, &apos;provider&apos;: &apos;python&apos;, &
↪apos;type&apos;: &apos;assert&apos;, &apos;_elapsed&apos;: 0.0003077983856201172}
{&apos;expression&apos;: &apos;0 == 0&apos;, &apos;provider&apos;: &apos;python&apos;,
↪ &apos;type&apos;: &apos;assert&apos;, &apos;_elapsed&apos;: 0.0002529621124267578}
</system-out></testcase></testsuite>
```

CHAPTER 14

Generate a custom JUnit XML report with custom properties and execution times metrics

You can track execution time metrics for monitoring and measure what is important to you. For example you can track using a machine interpretable format:

- response times (e.g., how much time is needed for returning a `POST` json payload)
- time that occurs between the invocation of an API and a reactive web application update or some asynchronous data appearing on an event store
- time that occurs between a user input on browser and results updated (e.g., a live search)
- time that occurs between a login button and the page loaded an usable (e.g., how much time is needed after a browser action to click on a target button)

14.1 Track response time metric in JUnit XML report

For example, a `test_categories.yml` file executed with the command line option `--junit-xml report.xml` (requires play_requests plugin):

```
test_data:
  - category: dev
  - category: movie
  - category: food
---
- type: GET
  provider: play_requests
  url: https://api.chucknorris.io/jokes/categories
  expression: "'$category' in response.json()"
- provider: metrics
  type: record_elapsed
  name: categories_time
- type: assert
  provider: python
```

(continues on next page)

(continued from previous page)

```
  expression: "variables['categories_time'] < 2.5"
  comment: you can make an assertion against the categories_time
```

will generate an extended `report.xml` file with custom properties like that:

```
<?xml version="1.0" encoding="utf-8"?><testsuite errors="0" failures="0" name="pytest
↪" skipped="0" tests="3" time="2.031"><testcase classname="test_categories.yml" file=
↪"test_categories.yml" name="test_categories.yml0" time="0.968"><properties>
↪<property name="categories_time" value="0.5829994678497314"/></properties><system-
↪out>{&apos;expression&apos;: &quot;&apos;dev&apos; in response.json()&quot;, &apos;
↪provider&apos;: &apos;play_requests&apos;, &apos;type&apos;: &apos;GET&apos;, &apos;
↪url&apos;: &apos;https://api.chucknorris.io/jokes/categories&apos;, &apos;_elapsed&
↪apos;: 0.5829994678497314}
{&apos;name&apos;: &apos;categories_time&apos;, &apos;provider&apos;: &apos;metrics&
↪apos;, &apos;type&apos;: &apos;record_elapsed&apos;, &apos;_elapsed&apos;: 3.
↪3855438232421875e-05}
{&apos;comment&apos;: &apos;you can make an assertion against the categories_time&
↪apos;, &apos;expression&apos;: &quot;variables[&apos;categories_time&apos;] &lt; 2.
↪5&quot;, &apos;provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;assert&
↪apos;, &apos;_elapsed&apos;: 0.0006382465362548828}
</system-out></testcase><testcase classname="test_categories.yml" file="test_
↪categories.yml" name="test_categories.yml1" time="0.481"><properties><property name=
↪"categories_time" value="0.4184422492980957"/></properties><system-out>{&apos;
↪expression&apos;: &quot;&apos;movie&apos; in response.json()&quot;, &apos;provider&
↪apos;: &apos;play_requests&apos;, &apos;type&apos;: &apos;GET&apos;, &apos;url&apos;
↪: &apos;https://api.chucknorris.io/jokes/categories&apos;, &apos;_elapsed&apos;: 0.
↪4184422492980957}
{&apos;name&apos;: &apos;categories_time&apos;, &apos;provider&apos;: &apos;metrics&
↪apos;, &apos;type&apos;: &apos;record_elapsed&apos;, &apos;_elapsed&apos;: 2.
↪09808349609375e-05}
{&apos;comment&apos;: &apos;you can make an assertion against the categories_time&
↪apos;, &apos;expression&apos;: &quot;variables[&apos;categories_time&apos;] &lt; 2.
↪5&quot;, &apos;provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;assert&
↪apos;, &apos;_elapsed&apos;: 0.000553131103515625}
</system-out></testcase><testcase classname="test_categories.yml" file="test_
↪categories.yml" name="test_categories.yml2" time="0.534"><properties><property name=
↪"categories_time" value="0.463592529296875"/></properties><system-out>{&apos;
↪expression&apos;: &quot;&apos;food&apos; in response.json()&quot;, &apos;provider&
↪apos;: &apos;play_requests&apos;, &apos;type&apos;: &apos;GET&apos;, &apos;url&apos;
↪: &apos;https://api.chucknorris.io/jokes/categories&apos;, &apos;_elapsed&apos;: 0.
↪463592529296875}
{&apos;name&apos;: &apos;categories_time&apos;, &apos;provider&apos;: &apos;metrics&
↪apos;, &apos;type&apos;: &apos;record_elapsed&apos;, &apos;_elapsed&apos;: 2.
↪09808349609375e-05}
{&apos;comment&apos;: &apos;you can make an assertion against the categories_time&
↪apos;, &apos;expression&apos;: &quot;variables[&apos;categories_time&apos;] &lt; 2.
↪5&quot;, &apos;provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;assert&
↪apos;, &apos;_elapsed&apos;: 0.00054931640625}
</system-out></testcase></testsuite>
```

and the custom property `categories_time` will be tracked for each test case execution, for example:

```
<properties>
    <property name="categories_time" value="0.5829994678497314"/>
</properties>
```

14.2 Advanced metrics in JUnit XML report

In this example we want to measures how long it takes a page to become interactive (page responding to user interactions) and evaluate update time for a live search feature. Let's see the `test_search.yml` example (requires play_selenium):

```
---
- provider: selenium
  type: get
  url: https://www.plone-demo.info/
- provider: metrics
  type: record_elapsed_start
  name: load_time
- provider: selenium
  type: setElementText
  text: plone 5
  locator:
    type: id
    value: searchGadget
- provider: metrics
  type: record_elapsed_stop
  name: load_time
- provider: metrics
  type: record_elapsed_start
  name: live_search_time
- provider: selenium
  type: waitForElementVisible
  locator:
    type: css
    value: li[data-url$="https://www.plone-demo.info/front-page"]
- provider: metrics
  type: record_elapsed_stop
  name: live_search_time
```

If you execute this scenario with the `--junit-xml results.xml` option you'll get a `results.xml` file similar to this one:

```
<?xml version="1.0" encoding="utf-8"?><testsuite errors="0" failures="0" name="pytest
↪" skipped="0" tests="1" time="13.650"><testcase classname="test_search.yml" file=
↪"test_search.yml" name="test_search.yml" time="13.580"><properties><property name=
↪"load_time" value="1.1175920963287354"/><property name="live_search_time" value="1.
↪0871295928955078"/></properties><system-out>{&apos;provider&apos;: &apos;selenium&
↪apos;, &apos;type&apos;: &apos;get&apos;, &apos;url&apos;: &apos;https://www.plone-
↪demo.info/&apos;, &apos;_elapsed&apos;: 9.593282461166382}
{&apos;name&apos;: &apos;load_time&apos;, &apos;provider&apos;: &apos;metrics&apos;, &
↪apos;type&apos;: &apos;record_elapsed_start&apos;, &apos;_elapsed&apos;: 1.
↪1682510375976562e-05}
{&apos;locator&apos;: {&apos;type&apos;: &apos;id&apos;, &apos;value&apos;: &apos;
↪searchGadget&apos;}, &apos;provider&apos;: &apos;selenium&apos;, &apos;text&apos;: &
↪apos;plone 5&apos;, &apos;type&apos;: &apos;setElementText&apos;, &apos;_elapsed&
↪apos;: 1.1019845008850098}
{&apos;name&apos;: &apos;load_time&apos;, &apos;provider&apos;: &apos;metrics&apos;, &
↪apos;type&apos;: &apos;record_elapsed_stop&apos;, &apos;_elapsed&apos;: 1.
↪9788742065429688e-05}
{&apos;name&apos;: &apos;live_search_time&apos;, &apos;provider&apos;: &apos;metrics&
↪apos;, &apos;type&apos;: &apos;record_elapsed_start&apos;, &apos;_elapsed&apos;: 1.
↪0013580322265625e-05}
```

(continues on next page)

(continued from previous page)

```
{&amp;apos;locator&amp;apos;: {&amp;apos;type&amp;apos;: &amp;apos;css&amp;apos;, &amp;apos;value&amp;apos;: &amp;apos;
↪li[data-url$=&amp;quot;https://www.plone-demo.info/front-page&amp;quot;]&amp;apos;}, &amp;apos;
↪provider&amp;apos;: &amp;apos;selenium&amp;apos;, &amp;apos;type&amp;apos;: &amp;apos;waitForElementVisible&
↪apos;, &amp;apos;_elapsed&amp;apos;: 1.060795545578003}
{&amp;apos;name&amp;apos;: &amp;apos;live_search_time&amp;apos;, &amp;apos;provider&amp;apos;: &amp;apos;metrics&
↪apos;, &amp;apos;type&amp;apos;: &amp;apos;record_elapsed_stop&amp;apos;, &amp;apos;_elapsed&amp;apos;: 2.
↪3603439331054688e-05}
</system-out></testcase></testsuite>
```

and in this case you'll find out that the key metric `load_time` was `1.11` seconds and the `live_search_time` was `1.09` seconds as you can see here:

```
<properties>
    <property name="load_time" value="1.1175920963287354"/>
    <property name="live_search_time" value="1.0871295928955078"/>
</properties>
```

So thanks to JUnit XML reporting you can track response times (not only browser based timings) using a machine readable format to be ingested by third party systems with an acceptable approximation if you cannot track timings directly on the systems under test.

14.3 Track any property in JUnit XML reports using expressions

Let's see a `test_categories.yml` (play_selenium required):

```
test_data:
  - category: dev
  - category: movie
  - category: food
---
- type: GET
  provider: play_requests
  url: https://api.chucknorris.io/jokes/categories
  expression: "'$category' in response.json()"
- provider: metrics
  type: record_property
  name: categories_time
  expression: "variables['_elapsed']*1000"
- type: assert
  provider: python
  expression: "variables['categories_time'] < 2500"
  comment: you can make an assertion against the categories_time
```

generates some custom properties (`categories_time` in milliseconds using a python expression) using the `--junit-xml results.xml` cli option:

```
<?xml version="1.0" encoding="utf-8"?><testsuite errors="0" failures="0" name="pytest
↪" skipped="0" tests="3" time="2.312"><testcase classname="test_categories.yml" file=
↪"test_categories.yml" name="test_categories.yml0" time="1.034"><properties>
↪<property name="categories_time" value="610.3124618530273"/></properties><system-
↪out>{&amp;apos;expression&amp;apos;: &amp;quot;&amp;apos;dev&amp;apos; in response.json()&amp;quot;, &amp;apos;
↪provider&amp;apos;: &amp;apos;play_requests&amp;apos;, &amp;apos;type&amp;apos;: &amp;apos;GET&amp;apos;, &amp;apos;
↪url&amp;apos;: &amp;apos;https://api.chucknorris.io/jokes/categories&amp;apos;, &amp;apos;_elapsed&
↪apos;: 0.6103124618530273}
```

(continues on next page)

(continued from previous page)


```
{&apos;expression&apos;: &quot;variables[&apos;_elapsed&apos;]*1000&quot;, &apos;
↪provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;exec&apos;, &apos;_
↪elapsed&apos;: 0.0006859302520751953}
{&apos;expression&apos;: &quot;variables[&apos;_elapsed&apos;]*1000&quot;, &apos;name&
↪apos;: &apos;categories_time&apos;, &apos;provider&apos;: &apos;metrics&apos;, &
↪apos;type&apos;: &apos;record_property&apos;, &apos;_elapsed&apos;: 0.
↪006484270095825195}
{&apos;comment&apos;: &apos;you can make an assertion against the categories_time&
↪apos;, &apos;expression&apos;: &quot;variables[&apos;categories_time&apos;] &lt;␣
↪2500&quot;, &apos;provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;
↪assert&apos;, &apos;_elapsed&apos;: 0.0005526542663574219}
</system-out></testcase><testcase classname="test_categories.yml" file="test_
↪categories.yml" name="test_categories.yml1" time="0.550"><properties><property name=
↪"categories_time" value="443.72105598449707"/></properties><system-out>{&apos;
↪expression&apos;: &quot;&apos;movie&apos; in response.json()&quot;, &apos;provider&
↪apos;: &apos;play_requests&apos;, &apos;type&apos;: &apos;GET&apos;, &apos;url&apos;
↪: &apos;https://api.chucknorris.io/jokes/categories&apos;, &apos;_elapsed&apos;: 0.
↪44372105598449707}
{&apos;expression&apos;: &quot;variables[&apos;_elapsed&apos;]*1000&quot;, &apos;
↪provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;exec&apos;, &apos;_
↪elapsed&apos;: 0.0009415149688720703}
{&apos;expression&apos;: &quot;variables[&apos;_elapsed&apos;]*1000&quot;, &apos;name&
↪apos;: &apos;categories_time&apos;, &apos;provider&apos;: &apos;metrics&apos;, &
↪apos;type&apos;: &apos;record_property&apos;, &apos;_elapsed&apos;: 0.
↪01613616943359375}
{&apos;comment&apos;: &apos;you can make an assertion against the categories_time&
↪apos;, &apos;expression&apos;: &quot;variables[&apos;categories_time&apos;] &lt;␣
↪2500&quot;, &apos;provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;
↪assert&apos;, &apos;_elapsed&apos;: 0.0011241436004638672}
</system-out></testcase><testcase classname="test_categories.yml" file="test_
↪categories.yml" name="test_categories.yml2" time="0.676"><properties><property name=
↪"categories_time" value="576.5485763549805"/></properties><system-out>{&apos;
↪expression&apos;: &quot;&apos;food&apos; in response.json()&quot;, &apos;provider&
↪apos;: &apos;play_requests&apos;, &apos;type&apos;: &apos;GET&apos;, &apos;url&apos;
↪: &apos;https://api.chucknorris.io/jokes/categories&apos;, &apos;_elapsed&apos;: 0.
↪5765485763549805}
{&apos;expression&apos;: &quot;variables[&apos;_elapsed&apos;]*1000&quot;, &apos;
↪provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;exec&apos;, &apos;_
↪elapsed&apos;: 0.0006375312805175781}
{&apos;expression&apos;: &quot;variables[&apos;_elapsed&apos;]*1000&quot;, &apos;name&
↪apos;: &apos;categories_time&apos;, &apos;provider&apos;: &apos;metrics&apos;, &
↪apos;type&apos;: &apos;record_property&apos;, &apos;_elapsed&apos;: 0.
↪006584644317626953}
{&apos;comment&apos;: &apos;you can make an assertion against the categories_time&
↪apos;, &apos;expression&apos;: &quot;variables[&apos;categories_time&apos;] &lt;␣
↪2500&quot;, &apos;provider&apos;: &apos;python&apos;, &apos;type&apos;: &apos;
↪assert&apos;, &apos;_elapsed&apos;: 0.0005452632904052734}
</system-out></testcase></testsuite>
```


obtaining the metrics you want to track for each execution, for example:

```
<properties><property name="categories_time" value="610.3124618530273"/></properties>
```

so you might track the category as well for each test execution or whatever you want.

CHAPTER 15

Monitoring test metrics with statsd/graphite

If you like the measure everything approach you can track and monitor interesting custom test metrics from an end user perspective during normal test executions or heavy load/stress tests thanks to the statsd/graphite integration.

Measuring important key metrics is important for many reasons:

- compare performance between different versions under same conditions using past tracked stats for the same metric (no more say the system *seems slower* today)
- predict the system behaviour with many items on frontend (e.g., evaluate the browser dealing with thousands and thousands of items managed by an infinite scroll plugin)
- predict the system behaviour under load

You can install `statsd/graphite` in minutes using Docker:

- https://graphite.readthedocs.io/en/latest/install.html

Basically you can track on `statsd/graphite` every **numeric** metric using the same commands used for tracking metrics on JUnit XML reports as we will see.

In addition, but not required, installing the third party plugin called pytest-statsd. you can track on `statsd/graphite`:

- execution times
- number of executed tests per status (pass, fail, error, etc)

Prerequisites (you need to install the optional statsd client not installed by default)::

```
pip install pytest-play[statsd]
```

Usage (cli options compatible with `pytest-statsd`):

```
--stats-d [--stats-prefix play --stats-host http://myserver.com --stats-port 3000]
```

where:

- `--stats-d`, enable `statsd`

- `--stats-prefix` (optional), if you plan on having multiple projects sending results to the same server. For example if you provide `play` as prefix you'll get a time metric under the `stats.timers.play.YOURMETRIC.mean` key (or instead of `.mean` you can use `.upper`, `upper_90`, etc)
- `--stats-host`, by default `localhost`
- `--stats-port`, by default `8125`

Now you can track timing metrics using the `record_elapsed` or `record_elapsed_start`/`record_elapsed_stop` commands seen before (pytest-play will send for you time values to `statsd` converted to `milliseconds` as requested by `statsd`).

If you want to track custom metrics using the `record_property` command you have to provide an additional parameter called `metric_type`. For example:

```
- provider: metrics
  type: record_property
  name: categories_time
  expression: "variables['_elapsed']*1000"
  metric_type: timing
- provider: metrics
  type: record_property
  name: fridge_temperature
  expression: "4"
  metric_type: gauge
```

Some additional information regarding the `record_property` command:

- if you don't provide the `metric_type` option in `record_property` commands values will not be transmitted to `statsd` (eventually they will be tracked on JUnit XML report if `--junit-xml` option was provided)
- if you provide an allowed `metric_type` value (`timing` or `gauge`) non numeric values will be considered as an error (`ValueError` exception raised)
- non allowed `metric_type` values will be considered as an error
- if you provide `timing` as `metric_type`, it's up to you providing a numeric value expressed in `milliseconds`

15.1 Monitor HTTP response times

Monitor API response time (see https://github.com/pytest-dev/pytest-play/tree/features/examples/statsd_graphite_monitoring):

15.2 Browser metrics

Monitor browser metrics using Selenium from an end user perspective (see https://github.com/pytest-dev/pytest-play/tree/features/examples/statsd_graphite_monitoring_selenium):

- from page load to page usable
- live search responsiveness

15.3 Record metrics programmatically

If you don't want to use `pytest-play` but you need to record test metrics you can use `pytest-play` as a library::

```
def test_programmatically(play):
    play.execute_command({
        'provider': 'metrics',
        'type': 'record_property',
        'name': 'oil_temperature',
        'expression': '60',
        'metric_type': 'gauge'})
```

CHAPTER 16

Performance tests with pytest-play and bzt/Taurus (BlazeMeter)

You can reuse all your pytest-play scenario and turn them to performance tests using bzt/Taurus (so it is compatible with BlazeMeter too and all its goodies).

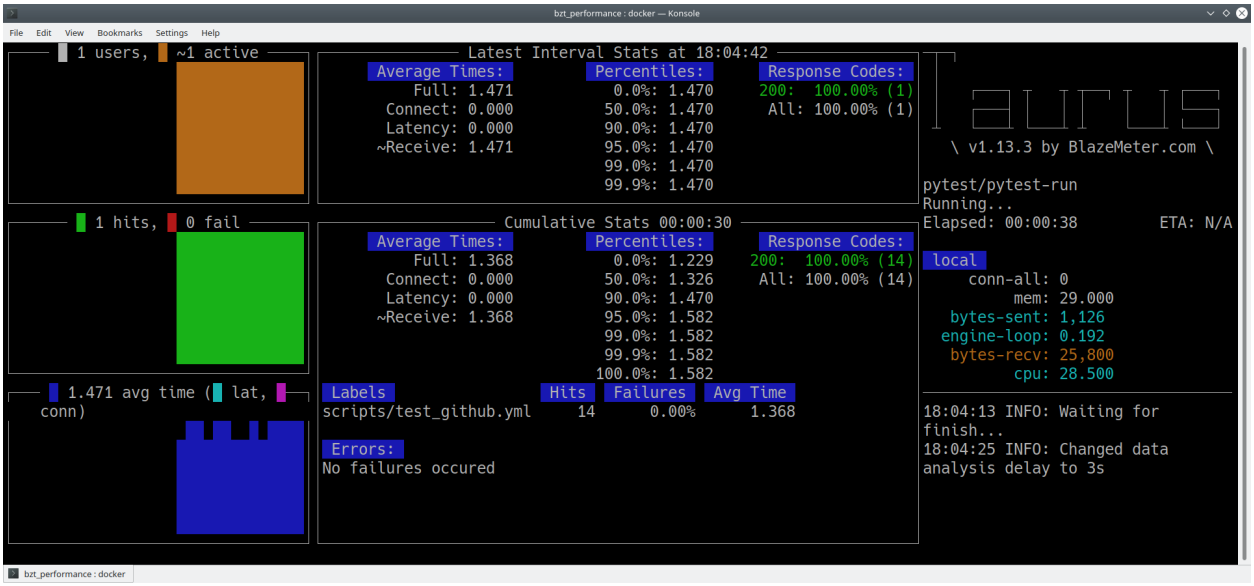
Add a bzt/Taurus YAML file with no `test_` prefix like that (full example here in bzt_performance):

```
settings:
  artifacts-dir: /tmp/%Y-%m-%d_%H-%M-%S.%f

execution:
- executor: pytest
  scenario: pytest-run
  iterations: 1

scenarios:
  pytest-run:
    # additional-args: --stats-d --stats-prefix play
    script: scripts/

services:
- module: shellexec
  prepare:
  - pip3 install -r https://raw.githubusercontent.com/davidemoro/pytest-play-docker/
↪master/requirements.txt
```

and run the following command:

```
docker run --rm -it -v $(pwd):/src --user root --entrypoint "bzt" davidemoro/pytest-
↪play bzt.yml
```

You will see bzt up and running playing our scenarios:



You can uncomment `additional-args` to pass other `pytest` command line options (e.g., enable `statsd` for key user metrics monitoring or any other cli option).

More info about bzt/Taurus here:

- http://gettaurus.org/

CHAPTER 17

Dynamic expressions in payloads without declaring variables

If you have to send a certain payload to a REST endpoint or a MQTT message containing a dynamic value you can store a variable with `store_variable` and use `$variable_name` in your payload when needed. Storing variables is cool if you will reuse later that value but if just have to generate a dynamic value, let's say a timestamp in milliseconds, you can use the `{! EXPRESSION !}` format.

For example (play_mqtt plugin required):

```
---
- comment: python expressions in mqtt payload (without declaring variables)
  provider: mqtt
  type: publish
  host: "$mqtt_host"
  port: "$mqtt_port"
  endpoint: "$mqtt_endpoint/$device_serial_number"
  payload: '{
        "measure_id":   [124],
        "obj_id_L":     [0],
        "measureType":  ["float"],
        "start_time":   {! int(datetime.datetime.utcnow().timestamp()*1000) !},
        "bin_value":    [1]
    }'
```

where instead of the expression:

```
{! int(datetime.datetime.utcnow().timestamp()*1000) !},
```

will be printed:

```
1553007973702
```

17.1 Browser based commands

The `pytest-play` core no more includes browser based commands. Moved to play_selenium external plugin.

17.2 pytest-play is pluggable and extensible

`pytest-play` has a pluggable architecture and you can extend it.

For example you might want to support your own commands, support non UI commands like making raw POST/GET/etc calls, simulate IoT devices activities, provide easy interaction with complex UI widgets like calendar widgets, send commands to a device using the serial port implementing a binary protocol and so on.

CHAPTER 18

How to register a new command provider

Let's suppose you want to extend pytest-play with the following command:

```
command = {'type': 'print', 'provider': 'newprovider', 'message': 'Hello, World!'}
```

You just have to implement a command provider:

```
from pytest_play.providers import BaseProvider

class NewProvider(BaseProvider):

    def this_is_not_a_command(self):
        """ Commands should be command_ prefixed """

    def command_print(self, command):
        print(command['message'])

    def command_yetAnotherCommand(self, command):
        print(command)
```

and register your new provider in your `setup.py` adding an entrypoint:

```
entry_points={
    'playcommands': [
        'print = your_package.providers:NewProvider',
    ],
},
```

You can define new providers also for non UI commands. For example publish MQTT messages simulating IoT device activities for integration tests.

If you want you can generate a new command provider thanks to:

- https://github.com/davidemoro/cookiecutter-play-plugin

18.1 Metadata format

You can also add some scenario metadata placing another YAML document on top of the scenario defined on the `test_XXX.yml` with the following format:

```
---
markers:
  - marker1
  - marker2
test_data:
  - username: foo
  - username: bar
---
# omitted scenario steps in this example...
```

Option details:

- `markers`, you can decorate your scenario with one or more markers. You can use them in pytest command line for filtering scenarios to be executed thanks to marker expressions like `-m "marker1 and not slow"`
- `test_data`, enables parametrization of your decoupletd test data and let you execute the same scenario many times. For example the example above will be executed twice (one time with "foo" username and another time with "bar")

New options will be added in the next feature (e.g., skip scenarios, xfail, xpass, etc).

18.2 Examples

- https://github.com/pytest-dev/pytest-play/tree/master/examples
- https://github.com/davidemoro/pytest-play-docker/tree/master/tests
- https://github.com/davidemoro/pytest-play-plone-example

18.3 Articles and talks

Articles:

- Hello pytest-play!
- API/REST testing like Chuck Norris with pytest play using YAML
- pytest-play automated docker hub publishing workflow
- Test automation framework thoughts and examples with Python, pytest and Jenkins

Talks:

- Serena Martinetti @ Pycon9 - Florence: Integration tests ready to use with pytest-play

18.4 Third party pytest-play plugins

- play_selenium, `pytest-play` plugin driving browsers using Selenium/Splinter under the hood. Selenium grid compatible and implicit auto wait actions for more robust scenarios with less controls.

- play_requests, `pytest-play` plugin driving the famous Python `requests` library for making `HTTP` calls.
- play_sql, `pytest-play` support for SQL expressions and assertions
- play_cassandra, `pytest-play` support for Cassandra expressions and assertions
- play_dynamodb, `pytest-play` support for AWS DynamoDB queries and assertions
- play_websocket, `pytest-play` support for websockets
- play_mqtt, `pytest-play` plugin for MQTT support. Thanks to `play_mqtt` you can test the integration between a mocked IoT device that sends commands on MQTT and a reactive web application with UI checks.

  You can also build a simulator that generates messages for you.

Feel free to add your own public plugins with a pull request!

18.5 Twitter

`pytest-play` tweets happens here:

- @davidemoro

CHAPTER 19

Welcome to pytest-play's documentation!

Contents:

19.1 Changelog

19.1.1 2.3.0 (2019-04-05)

Features and improvements:

- `wait_until` and `wait_until_not` now accept commands with no `sub_commands` property
- implement new `while` command in python provider (while expression is true)

19.1.2 2.2.2 (2019-03-29)

Minor changes:

- remove internal property parameter on engine

Bugfix:

- add compatibility with `pytest-repeat`'s `--count` command line option

Documentation:

- mention how to generate dynamic values using `{! expr !}` expressions (e.g., dynamic payloads in REST or MQTT without having to store variables when not needed)

19.1.3 2.2.1 (2019-03-19)

Minor changes:

- add `int` and `float` builtins available in Python expressions

- make python expressions more flexible for future improvements (internal change that doesn't affect compatibility)

Bugfix:

- fix `--setup-plan` invokation

Documentation:

- add more examples (bzt/Taurus and performance tests using pytest-play)

19.1.4 2.2.0 (2019-03-01)

- `statsd` integration (optional requirement) for advanced test metrics using statsd/graphite. If you install pytest play with the optional statsd support with `pytest-play[statsd]` you will get the additional dependency `statsd` client and you can use the same cli options defined by the `pytest-statsd` plugin (e.g., `--stats-d [--stats-prefix myproject --stats-host http://myserver.com --stats-port 3000]`).

  Note well: despite the above cli options are the same defined by the `pytest-statsd` plugin, at this time of writing `pytest-statsd` is not a `pytest-play` dependency so you won't get stats about number of failures, passing, etc but only stats tracked by `pytest-play`. If you need them you can install `pytest-statsd` (it plays well with `pytest-play`)

19.1.5 2.1.0 (2019-02-22)

Features:

- support junit xml generation file with `system-out` element for each test case execution (pytest `--junit-xml` option). `system-out` will tracked by default in junit report unless you use the `--capture=no` or its alias `-s`
- track `_elapsed` time for each executed command `--junit-xml` report if `system-out` is enabled
- track `pytest` custom properties in `--junit-xml` report for monitoring and measure what is important to you. For example you can track as key metric the time of the time occurred between the end of the previous action and the completion of the following. Basically you can track under the `property_name` *load_login* key the time occurred between the click on the submit button and the end of the current command (e.g., click on the menu or text input being able to receive text) using a machine interpretable format.

  The `property_name` value elapsed time will be available as standard `pytest-play` variable so that you can make additional assertions
- after every command execution a `pytest-play` variable will be added/updated reporting the elapsed time (accessible using `variables['_elapsed']`).

  So be aware that the `_elapsed` variable name should be considered as a special variable and so you should not use this name for storing variables
- improve debug in case of failed assertions or errored commands. Logged variables dump in standard logs and `system-out` reporting if available
- improve debuggability in case of assertion errors (log failing expression)
- added a new `metrics` provider that let you track custom metrics in conjunction with `--junit-xml` option. You can track in a machine readable format response times, dynamic custom expressions, time that occurs between different commands (e.g., measure the time needed after a login to interact with the page, time before an asynchronous update happens and so on). Under the `metrics` provider you'll find the `record_property`, `record_elapsed`, `record_elapsed_start` and `record_elapsed_stop` commands

Documentation:

- minor documentation changes
- add more examples

19.1.6 2.0.2 (2019-02-06)

Documentation:

- more examples
- fix documentation bug on README (example based on selenium with missing `provider: selenium`)

19.1.7 2.0.1 (2019-01-30)

Documentation:

- Mention davidemoro/pytest-play docker container in README. You can use pytest-play with a docker command like that now `docker run -i --rm -v $(pwd):/src davidemoro/pytest-play`

Bugfix:

- Fix error locking pipenv due to pytest-play requirement constraint not existing (RestrictedPython>=4.0.b2 -> RestrictedPython>=4.0b2)

19.1.8 2.0.0 (2019-01-25)

Breaking changes:

- Renamed fixture from *play_json* to *play* (#5)
- Drop json support, adopt yaml only format for scenarios (#5)
- Drop `.ini` file for metadata, if you need them you can add a YAML document on top of the scenario `.yml` file. You no more need multiple files for decorating your scenarios now (#65)
- *play.execute* no more accepts raw data string), consumes a list of commands. Introduced *play.execute_raw* accepting raw data string.
- *play.execute_command* accepts a Python dictionary only now (not a string)
- Selenium provider removed from `pytest-play` core, implemented on a separate package `play_selenium`. Starting from now you have to add to your selenium commands `provider: selenium`
- engine's `parametrizer_class` attribute no more available (use `parametrizer.Parametrizer` by default now)

Bug fix:

- Fix invalid markup on PyPI (#55)
- Fix invalid escape sequences (#62).

Documentation and trivial changes:

- Add examples folder

19.1.9 1.4.2 (2018-05-17)

- Configuration change on Github. Use the same branching policy adopted by pytest (master becomes main branch, see #56)
- Fixed skipped test and added new tests (deselect scenarios with keyword and marker expressions)
- Fix #58: you no more get a TypeError if you try to launch pytest-play in autodiscovery mode
- Fix #55: restructured text lint on README.rst (bad visualization on pypi)
- Updated README (articles and talks links)
- Added a `DeprecationWarning` for *play_json* fixture. pytest-play will be based on yaml instead of json in version >=2.0.0. See https://github.com/pytest-dev/pytest-play/issues/5

19.1.10 1.4.1 (2018-04-06)

- Documentation improvements
- Add bzt/Taurus/BlazeMeter compatibility

19.1.11 1.4.0 (2018-04-05)

- Small documentation improvements
- Now `test_XXX.json` files are automatically collected and executed
- You can run a test scenario using the pytest CLI `pytest test_YYY.json`
- Introduced json test scenario ini file with markers definition. For a given `test_YYY.json` scenario you can add a `test_YYY.ini` ini file:

```
[pytest]
markers =
    marker1
    marker2
```

  and filter scenarios using marker expressions `pytest -m marker1`

- Enabled parametrization of arguments for a plain json scenario in scenario ini file:

```
[pytest]
test_data =
   {"username": "foo"}
   {"username": "bar"}
```

  and your json scenario will be executed twice

- `pytest-play` loads some variables based on the contents of the optional `pytest-play` section in your `pytest-variables` file now. So if your variables file contains the following values:

```
pytest-play:
  foo: bar
  date_format: YYYYMMDD
```

  you will be able to use expressions `$foo`, `$date_format`, `variables['foo']` or `variables['date_format']`

19.1.12 1.3.2 (2018-02-05)

- Add `sorted` in python expressions

19.1.13 1.3.1 (2018-01-31)

- Add more tests
- Documentation update
- play_json fixture no more assumes that you have some pytest-variables settings. No more mandatory
- fix include scenario bug that occurs only on Windows (slash vs backslash and JSON decoding issues)

19.1.14 1.3.0 (2018-01-22)

- documentation improvements
- supports teardown callbacks

19.1.15 1.2.0 (2018-01-22)

- implement python based commands in `pytest-play` and deprecates `play_python`. So this feature is a drop-in replacement for the `play-python` plugin.

  You should no more install `play_python` since now.
- update documentation
- deprecate selenium commands (they will be implemented on a separate plugin and dropped in `pytest-play` >= 2.0.0). All your previous scripts will work fine, this warning is just for people directly importing the provider for some reason.
- implement skip conditions. You can omit the execution of any command evaluating a Python based skip condition

19.1.16 1.1.0 (2018-01-16)

- Documentation updated (add new pytest play plugins)
- Support default payloads for command providers. Useful for HTTP authentication headers, common database settings

19.1.17 1.0.0 (2018-01-10)

- execute command accepts kwargs now
- execute command returns the command value now
- complete refactor of `include` provider (no backwards compatibility)
- add `play_json.get_file_contents` and removed `data_getter` fixture (no backwards compatibility)

19.1.18 0.3.1 (2018-01-04)

- play engine now logs commands to be executed and errors

19.1.19 0.3.0 (2018-01-04)

- you are able to update variables when executing commands
- you can extend `pytest-play` with new pluggable commands coming from third party packages thanks to setuptools entrypoints

19.1.20 0.2.0 (2018-01-02)

- no more open browser by default pytest-play is a generic test engine and it could be used for non UI tests too.

  So there is no need to open the browser for non UI tests (eg: API tests)

19.1.21 0.1.0 (2017-12-22)

- implement reusable steps (include scenario)
- minor documentation changes

19.1.22 0.0.1 (2017-12-20)

- First release

19.2 API

Here you can see the technical documentation.

19.2.1 pytest_play.plugin

pytest_play.plugin.**pytest_addoption**(*parser*)

Parameters **parser** –

Returns

pytest_play.plugin.**pytest_collect_file**(*parent*, *path*)
Collect test_XXX.yml files

class pytest_play.plugin.**YAMLFile**(*fspath*, *parent=None*, *config=None*, *session=None*, *nodeid=None*)

collect()
returns a list of children (items and collectors) for this collection node.

class pytest_play.plugin.**YAMLItem**(*name*, *parent=None*, *config=None*, *session=None*, *nodeid=None*, *callspec=None*, *keywords=None*, *originalname=None*)

module
Needed for Taurus/bzt/BlazeMeter compatibility See https://bit.ly/2GE2KS4

pytest_play.plugin.**play_engine_class**(**args*, ***kwargs*)
Play engine class class

pytest_play.plugin.**play**(**args*, ***kwargs*)
How to use yml_executor:

```
def test_experimental(play):
    data = play.get_file_contents(
        '/my/path/etc', 'login.yml')
    play.execute_raw(data)
```

19.2.2 pytest_play.executors

CHAPTER 20

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index