pytest-monitor

Release pytest-monitor v1.3.0

Jean-Sébastien Dieu

May 13, 2020

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

pytest-monitor tracks the resources (like memory and compute time) consumed by a test suite, so that you can make sure that your code does not use too much of them.

Thanks to *pytest-monitor*, you can check resource consumption in particular through continuous integration, as this is done by monitoring the consumption of test functions. These tests can be functional (as usual) or be dedicated to the resource consumption checks.

1.1 Use cases

Examples of use cases include technical stack updates, and code evolutions.

1.1.1 Technical stack updates

In the Python world, libraries often depends on several packages. By updating some (or all) of the dependencies, you update code that you do not own and therefore do not control. Tracking your application's resource footprint can prevent unwanted resource consumption, and can thus validate the versions of the packages that you depend on.

1.1.2 Code evolution

Extending your application with new features, or fixing its bugs, might have an impact on the core of your program. The performance of large applications or libraries can be difficult to assess, but by monitoring resource consumption, *pytest-monitor* allows you to check that despite code udpates, the performance of your code remains within desirable limits.

1.2 Example

```python
import pytest
import time


# Tests are run and monitored by default: no boilerplate code needed
def test_sleep1():
    time.sleep(1)

# Run as a test, but do not monitor:
@pytest.mark.monitor_skip_test
```

(continues on next page)

(continued from previous page)

```
def test_sleep2():
    time.sleep(2)

# Support for parametrized tests (monitored by default):
@pytest.mark.parametrize(('range_max', 'other'), [(10, "10"), (100, "100"), (1000,
↪"1000"), (10000, "10000")])
def test_heavy(range_max, other):
    assert len(['a' * i for i in range(range_max)]) == range_max
```

CHAPTER TWO

INSTALLATION

pytest-monitor is a plugin for *pytest*.

2.1 Supported environments

pytest-monitor currently works on *Linux* and *macOS*. Support for *Windows* is experimental and not tested.

You will need pytest 4.4+ to run pytest-monitor.

The following versions of Python are supported:

- Python 3.5
- Python 3.6
- Python 3.7

Support for Python 3.8 is still experimental.

2.2 From conda

Simply run the following command to get it installed in your current environment

```
conda install pytest-monitor -c https://conda.anaconda.org/conda-forge
```

2.3 From pip

Simply run the following command to get it installed

```
pip install pytest-monitor
```

CHAPTER

THREE

CONFIGURING YOUR SESSION

pytest-monitor gives you flexibility for running your test suite. In this section, we will discuss the different available options, and how they influence the *pytest* session.

3.1 Scope Restriction

pytest-monitor is able to restrict the scope of the analysis. As a default, only tests functions discovered by pytest are monitored.

Sometime, you might want to monitor a whole module or test session. This can be achieved thanks to the *–restrict-scope-to* option.

If a scope restriction is set, then the monitoring will be performed at the selected levels. For example, monitoring at both function and module level can be achieved by the following command:

```
pytest --restrict-scope-to function,module
```

Accepted values are:

- function: test functions will be monitored individually, leading to one entry per test function.
- module: each discovered module will be monitored regardless of the others.
- class: test class objects will be monitored individually.
- session: monitor the whole session.

It is important to realize that using multiple scopes has an impact on the monitoring measures. For example, the *pytest-monitor* code that monitors functions does consume resources for each function (notably compute time). As a consequence, the resources consumed by their module will include the resources consumed by *pytest-monitor* for each function. If individual functions were not monitored, the resource consumption reported for the module would therefore be lower.

Due to the way *pytest* handles test modules, some specificities apply when monitoring modules:

- The total measured elapsed time includes the setup/teardown process for each function. On the other hand, a function object measures only the duration of the function run (without the setup and teardown parts).
- Consumed memory will be the peak of memory usage during the whole module run.

3.2 Handling parameterized tests

Parameterized tests can be introspected by *pytest-monitor* during the setup phase: their real name is based on the parameter values. This uses the string representation of the parameters (so you want to make sure that this representation suits your needs).

Let's consider the following test:

```
@pytest.mark.parametrize(('asint', 'asstr'), [(10, "10"), (100, "100"), (1000, "1000
↪"), (10000, "10000")])
def test_p(asint, asstr):
    assert asint == int(asstr)
```

By default, *pytest-monitor* will generate the following entries:

- test_p[10-10]
- test_p[100-100]
- test_p[1000-1000]
- test_p[10000-10000]

You can ask *pytest-monitor* to tag parameters with their names (as provided by `@pytest.mark.parametrize`), with the following option:

```
pytest --parametrization-explicit
```

which will lead to the following entries:

- test_p[asint_10-asstr_10]
- test_p[asint_100-asstr_100]
- test_p[asint_1000-asstr_1000]
- test_p[asint_10000-asstr_10000]

3.3 Disable monitoring

If you need for some reason to disable the monitoring, pass the *–no-trace* option.

3.4 Adding a description to a run

Sometimes, you might want to compare identical state of your code. In such cases, relying only on the scm references and the run date of the session. For that, *pytest-monitor* can assist you by adding a description field to your session. Setting a description is as simple as this:

```
bash $> pytest --description "Any run description you want"
```

CHAPTER

FOUR

MANAGING YOUR TEST SUITE

pytest-monitor does not require any specific setup: it is active by default. Thus all your tests are by default analyzed in order to collect monitored information.

4.1 About collecting and storing results

pytest-monitor makes a clear distinction between the execution context and the test metrics. This distinction can been seen clearly in the code and the initialization sequence:

1. Collect environment values. Various pieces of information about the machine are collected.
2. Store the context. The Execution Context collected in step #1 is recorded if not yet known.
3. Prepare the run. In order to provide more accurate measurements, we "warm up" the context and take an initial set of measurements. Some will be used for adjusting later measurements.
4. Run tests and enable measurements. Depending on the item type (function, class or module), we launch the relevant measurements. Each time a monitored item ends, the measurement results (Metrics) are recorded right away.
5. End session. If sending the monitoring results to a remote server has been requested, this is when *pytest-monitor* does it.

4.2 Selecting tests to monitor

By default, all tests are monitored, even small ones which would not require any specific monitoring. It is possible to control more finely which tests will be monitored by *pytest-monitor*. This is done through the use of *pytest* markers.

pytest-monitor offers two markers for this:

`@pytest.mark.monitor_skip_test` marks your test for execution, but without any monitoring.

`@pytest.mark.monitor_skip_test_if(cond)` tells *pytest-monitor* to execute the test but to monitor results if and only if the condition is true.

Here is an example:

```python
import pytest
import sys


def test_execute_and_monitor():
    assert True
```

(continues on next page)

(continued from previous page)

```python

@pytest.mark.monitor_skip_test
def test_execute_do_not_monitor():
    assert True

@pytest.mark.monitor_skip_test_if(sys.version_info >= (3,))
def test_execute_and_monitor_py3_or_above():
    assert True
```

4.3 Disabling monitoring except for some tests

pytest offers global markers. For example, one can set the default to no monitoring:

```python
import pytest

# With the following global module marker,
# monitoring is disabled by default:
pytestmark = [pytest.mark.monitor_skip_test]
```

In this case, it is necessary to explicitly activate individual monitoring. This is accomplished with:

`@pytest.mark.monitor_test` marks your test as to be executed and monitored, even if monitoring is disabled for the module.

`@pytest.mark.monitor_test_if(cond)` tells *pytest-monitor* to execute the test and to monitor results if and only if the condition is true, regardless of the module monitor setup.

Continuing the example above:

```python
import time
import sys


def test_executed_not_monitored():
    time.sleep(1)
    assert True

def test_executed_not_monitored_2():
    time.sleep(2)
    assert True

@pytest.mark.monitor_test
def test_executed_and_monitored():
    assert True

@pytest.mark.monitor_test_if(sys.version_info >= (3, 7))
def test_executed_and_monitored_if_py37():
    assert True
```

4.4 Associating your tests to a component

pytest-monitor allows you to *tag* each test in the database with a "**component**" name. This allows you to identify easily tests that come from a specific part of your application, or for distinguishing test results for two different projects that use the same *pytest-monitor* database.

Setting up a component name can be done at module level:

```
import time
import pytest


pytest_monitor_component = "my_component"  # Component name stored in the results␣
↪database

def test_monitored():
    t_a = time.time()
    b_continue = True
    while b_continue:
        t_delta = time.time() - t_a
        b_continue = t_delta < 1
    assert not b_continue
```

If no *pytest_monitor_component* variable is defined, the component is set to the empty string. In projects with many modules, this can be tedious. *pytest-monitor* therefore allows you to force a fixed component name for the all the tests:

```
$ pytest --force-component YOUR_COMPONENT_NAME
```

This will force the component value to be set to the one you provided, whatever the value of *pytest_monitor_component* in your test module, if any.

If you need to use a global component name for all your tests while allowing some modules to have a specific component name, you can ask *pytest-monitor* to add a prefix to any module-level component name:

```
$ pytest --component-prefix YOUR_COMPONENT_NAME
```

This way, all tests detected by *pytest* will have their component prefixed with the given value (tests for modules with no *pytest_monitor_component* variable are simply tagged with the prefix).

For instance the following test module:

```
import time
import pytest


pytest_monitor_component = "component_A"

def test_monitored():
    t_a = time.time()
    b_continue = True
    while b_continue:
        t_delta = time.time() - t_a
        b_continue = t_delta < 1
    assert not b_continue
```

will yield the following value for the component fields, depending on the chosen command-line option:

Command line used	Component value
pytest –force-component PROJECT_A	PROJECT_A
pytest –component-prefix PROJECT_A	PROJECT_A.component_A

CHAPTER

FIVE

EXPLOITING MEASURES

5.1 Storage

Once measures are collected, *pytest-monitor* dumps them either in a local database or sends them to a monitor server.

In the case of local storage, a *sqlite3* database is used, as it is lightweight and is provided with many Python distributions (being part of the standard library).

Measures are stored in the *pytest* invocation directory, in a database file named **.pymon**. You are free to override the name of this database by setting the *–db* option:

```
pytest --db /path/to/your/monitor/database
```

You can also sends your tests result to a monitor server (under development at that time) in order to centralize your Metrics and Execution Context (see below):

```
pytest --remote server:port
```

5.2 Execution Context, Metrics and Session

We distinguish two kinds of measures:

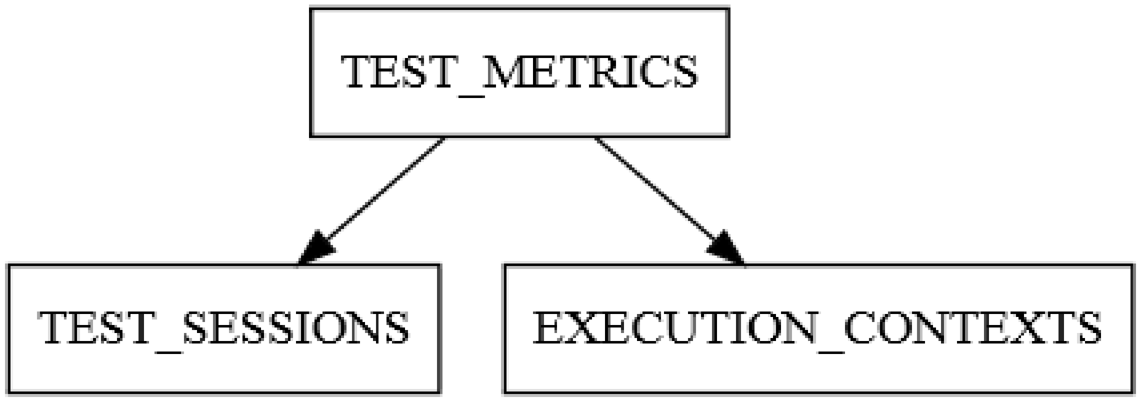
- those related to the **Execution Context**. This is related to your machine (node name, CPU, memory…),
- the **Metrics** related to the tests themselves (this can be the memory used, the CPU usage…).

Regarding tests related **metrics**, one can see metrics which are tests independent and those which are session independent (session start date, scm reference). For this reason, *pytest-monitor* uses a notion of session metrics to which each tests are linked to.

Additionally, each test is linked to an Execution Context so that comparisons between runs is possible.

5.3 Model

The local database associates each test Metrics to the specific context in which it was run:



5.3.1 Execution Context

Execution Contexts are computed prior to the start of the *pytest* session. An Execution Context describes much of the machine settings:

CPU_COUNT (integer) Number of online CPUs the machine can use.

CPU_FREQUENCY_MHZ (integer) Base frequency of the CPUs (in megahertz).

CPU_VENDOR (TEXT 256 CHAR) Full CPU vendor string.

RAM_TOTAL_MB (INTEGER) Total usable RAM (physical memory) in megabytes.

MACHINE_NODE (TEXT 512 CHAR) Fully qualified domain name of the machine.

MACHINE_TYPE (TEXT 32 CHAR) Machine type.

MACHINE_ARCH (TEXT 16 CHAR) Mode used (64 bits…).

SYSTEM_INFO (TEXT 256 CHAR) Operating system name and release level.

PYTHON_INFO (TEXT 512 CHAR) Python information (version, compilation mode used and so on…)

ENV_H (TEXT 64 CHAR) Hash string used to uniquely identify an execution context.

In the local database, Execution Contexts are stored in table *EXECUTION_CONTEXTS*.

5.4 Sessions

SESSION_H (TEXT 64 CHAR) Hash string used to uniquely identify a session run.

RUN_DATE (TEXT 64 CHAR) Time at which the *pytest* session was started. The full format is 'YYYY-MM-DDTHH:MM:SS.uuuuuu' (ISO 8601 format with UTC time). The fractional second part is omitted if it is zero.

SCM_ID (TEXT 128 CHAR) Full reference to the source code management system if any.

RUN_DESCRIPTION (TEXT 1024 CHAR) A free text field that you can use to describe a session run.

In the local database, Sessions are stored under the table *TEST_SESSIONS*.

5.4.1 Metrics

Metrics are collected at test, class and/or module level. For both classes and modules, some of the metrics can be skewed due to the technical limitations described earlier.

SESSION_H (TEXT 64 CHAR) Session context used for this test.

ENV_H (TEXT 64 CHAR) Execution Context used for this test.

ITEM_START_TIME (TEXT 64 CHAR) Time at which the item test was launched. The full format is 'YYYY-MM-DDTHH:MM:SS.uuuuuu' (ISO 8601 format with UTC time). The fractional second part is omitted if it is zero.

ITEM_PATH (TEXT 4096 CHAR) Path of the item, using an import compatible string specification.

ITEM (TEXT 2096 CHAR) Initial item name, without any variant.

ITEM_VARIANT varchar(2048) Full item name, with parametrization used if any.

ITEM_FS_LOC varchar(2048) Item's module path relative to pytest invocation directory.

KIND (TEXT 64 CHAR) Type of item (function, class, module. . .).

COMPONENT (TEXT 512 CHAR), NULLABLE Component to which the test belongs, if any (this is used when sending results to a server, for identifying each source of Metrics).

TOTAL_TIME (FLOAT) Total time spent running the item (in seconds).

USER_TIME (FLOAT) Time spent in User mode (in seconds).

KERNEL_TIME (FLOAT) Time spent in Kernel mode (in seconds).

CPU_USAGE (FLOAT) System-wide CPU usage as a percentage (100 % is equivalent to one core).

MEM_USAGE (FLOAT) Maximum resident memory used during the test execution (in megabytes).

In the local database, these Metrics are stored in table *TEST_METRICS*.

CHAPTER

SIX

USE OF A REMOTE SERVER

You can easily send your metrics to a remote server. This can turn usefull when it comes to running tests in parallel with plugins such as *pytest-xdist* of *pytest-parallel*. To do so, instruct pytest with the remote server address to use:

```
bash $> pytest --remote myremote.server.net:port
```

This way, *pytest-monitor* will automatically send and query the remote server as soon as it gets a need. Note that *pytest-monitor* will revert to a normal behaviour if:

- it cannot query the context or the session for existence
- it cannot create a new context or a new session

CHAPTER

SEVEN

IMPLEMENTING A REMOTE SERVER

7.1 How pytest-monitor interacts with a remote server

The following sequence is used by *pytest-monitor* when using a remote server:

1. Ask the remote server if the **Execution Context** is known.
2. Insert the **Execution Context** if the server knows nothing about it.
3. Ask the remote server if the **Session** is known.
4. Insert the **Session** if the server knows nothing about it.
5. Insert results once measures have been collected.

7.2 Used HTTP codes

Two codes are used by *pytest-monitor* when asked to work with a remote server:

- 200 (OK) is used to indicate that a query has led to a non-empty result.
- 201 (CREATED) is expected by *pytest-monitor** when sending a new entry (**Execution Context**, **Session** or any **Metric**).
- 204 (NO CONTENT) though not checked explicitely should be returned when a request leads to no results.

7.3 Mandatory routes

The following routes are expected to be reachable:

GET /contexts/<str:hash>

Query the system for a **Execution Context** with the given hash.

Return Codes: Must return *200* (*OK*) if the **Execution Context** exists, *204* (*NO CONTENT*) otherwise

GET /sessions/<str:hash>

Query the system for a **Session** with the given hash.

Return Codes: Must return *200* (*OK*) if the **Session** exists, *204* (*NO CONTENT*) otherwise

POST /contexts/

Request the system to create a new entry for the given **Execution Context**. Data are sent using Json parameters:

```
{
    cpu_count: int,
    cpu_frequency: int,
    cpu_type: str,
    cpu_vendor: str,
    ram_tota: int,
    machine_node: str,
    machine_type: str,
    machine_arch: str,
    system_info: str,
    python_info: str,
    h: str
}
```

Return Codes: Must return *201* (*CREATED*) if the **Execution Context** has been created

POST /sessions/

Request the system to create a new entry for the given **Session**. Data are sent using Json parameters:

```
{
    session_h: str,
    run_date: str,
    scm_ref: str,
    description: str
}
```

Return Codes: Must return *201* (*CREATED*) if the **Session** has been created

POST /metrics/

Request the system to create a new **Metrics** entry. Data are sent using Json parameters:

```
{
    session_h: str,
    context_h: str,
    item_start_time: str,
    item_path: str,
    item: str,
    item_variant: str,
    item_fs_loc: str,
    kind: str,
    component: str,
    total_time: float,
    user_time: float,
    kernel_time: float,
    cpu_usage: float,
    mem_usage: float
}
```

Return Codes: Must return *201* (*CREATED*) if the **Metrics** has been created

CHAPTER

EIGHT

CONTRIBUTION GUIDE

If you want to contribute to this project, you are welcome to do so!

8.1 Create your own development environment

We use conda as our main packaging system, though pip works as well.

The following instructions describe how to create your development environment using conda:

1. Create a new environment:

   ```
   conda create -n pytest-monitor-dev python=3 -c https://conda.anaconda.org/
   ↪conda-forge -c defaults
   ```

2. Install the dependencies:

   ```
   conda install --file requirements.txt -n pytest-monitor-dev -c https://
   ↪conda.anaconda.org/conda-forge -c defaults
   ```

3. Activate your environment:

   ```
   conda activate pytest-monitor-dev
   ```

4. Install *pytest-monitor* in development mode:

   ```
   python setup.py develop
   ```

5. You're done!

8.2 Feature requests and feedback

We would be happy to hear about your propositions and suggestions. Feel free to submit them as issues and:

- Explain in details the expected behavior.
- Keep the scope as narrow as possible. This will make them easier to implement.

8.3 Bug reporting

Report bugs for *pytest-monitor* in the issue tracker. Every filed bugs should include:

- Your operating system name and version.
- **Any details about your local setup that might be helpful in troubleshooting, specifically:**
    - the Python interpreter version,
    - installed libraries,
    - and your *pytest* version.
- Detailed steps to reproduce the bug.

8.4 Bug fixing

Look through the GitHub issues for bugs. Talk to developers to find out how you can fix specific bugs.

8.5 Feature implementation

Look through the GitHub issues for enhancements.

Talk to developers to find out how you can implement specific features.

Thank you!

CHANGELOG

- : Normalized http codes used for sending metrics to a remote server.
- : Change default analysis scope to function.
- : A local database is always created even with –no-db option passed.
- : No execution contexts pushed when using a remote server.
- : Fix remote server interface for sending measures.
- : Added an option to add a description to a pytest run
- : Compute user time and kernel time on a per test basis for clarity and ease of exploitation.
- : Extend item information and separate item from its variants.
- : pytest-monitor hangs infinitely when a pytest outcome (skip, fail. . .) is issued.
- : Initial release

CHAPTER TEN

INDICES AND TABLES

- genindex
- modindex
- search