# metric-farmer Documentation

team useblocks

Aug 27, 2019

# Contents:

**Cares about your project metrics**

Metric-Farmer is a Python based command line application to collect and store metrics from different sources to different targets.

It is designed to easily create and maintain complex metric measurements and to painless integrate its functions into continuous integration systems (CI/CD) or local task executions (cron jobs).

The configuration is completely done by JSON-based `farm`-files in a project folder called `.farmer`. And no development skills are needed to use Metric-Farmer.

Developers can easily create own Metric-Farmer extensions to provide custom solutions for sources and targets. For instance to measure a company-internal service.

# CHAPTER 1

# Features

Metric-Farmer is capable to collect metrics from following *Sources*:

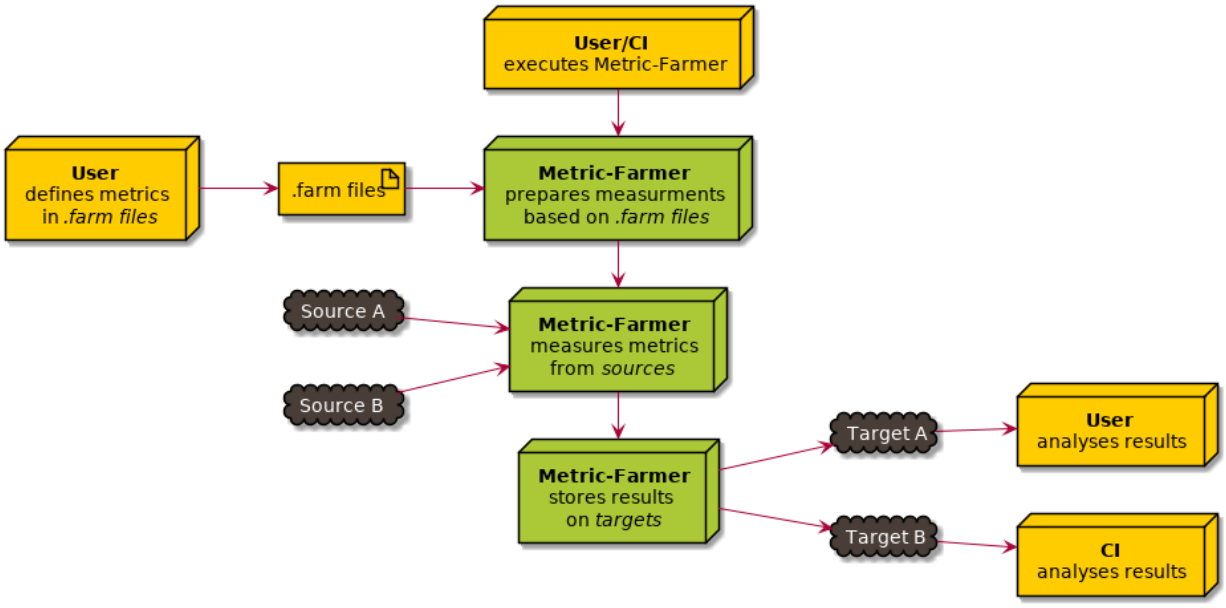
- *static values*
- *random values*
- *file count*
- *REST requests* (E.g. to measure *JIRA* or *GitHub*)

And sends the results to the following *Targets*:

- print output (*text* or *json*)
- file output (*text* or *json*)
- database output (*sqlite*)

CHAPTER 2

# Workflow



CHAPTER 3

# Quick start

## 3.1 Installation

You need to have an installed Python 3.5 or above environment.

Then install Metric-Farmer by executing `pip install metricfarmer`. A working internet connection is needed!

## 3.2 First measurements

Execute metricfarmer by following command on your command prompt:

```
metricfarmer
```

You will see that extensions and configurations got loaded, but no metrics have been measured. That's because you haven't defined them yet.

Luckily Metric-Farmer comes with some metric examples, which can be used by asking for all metrics with the tag `mf_examples`.

So simply execute:

```
metricfarmer --tags mf_examples print
```

`print` is a predefined target and prints out the results of all measured metrics.

## 3.3 Playing with targets

Targets define what shall happen with the measured metric results.

A very basic target is the `print` target. Other predefined targets are `print_json`, `file_text` and `file_json`.

You are free to combine them during your call:

```
metricfarmer -t mf_examples print file_json
```

`-t` is an abbreviation of `--tags` and `file_json` will print the results in a file called `metric_results.json` on your current working directory.

Take a look into *Targets* to know how to define easily own, custom targets for your special use cases.

## 3.4 Knowing what is possible

To get a list of all available metrics, tags, sources, targets and more, simply execute:

```
metricfarmer --list
```

## 3.5 Own metrics

Metric-Farmer is controlled completely by file-based configurations. Therefore it looks for `.farm` files at the following locations:

1. Metric-Farmer installation folder (for basic and example configs)
2. `.farmer` folder in user/home directory (e.g. ~/.farmer)
3. `.farmer` folder in current working directory (normally your project root)

Metric-Farmer reads in **all** found `.farm`-files and combines them to a single configuration object.

Later read configuration parameters overrides previous read configurations. So a project-configuration overrides always configurations coming from the user/home folder.

For a simple example, execute the following steps:

1. Create a folder `.farmer` on your project root folder
2. Create a file inside `.farmer` called `my_metrics.farm`
3. Add the following content to the file:

```
{
   "metrics": {
      "my_metric": {
         "source": {
            "type": "random",
         }
      }
   }
}
```

4. On your project root level execute `metricfarmer print`.

Congratulations, you have created and measured your first own metric. As we have used the predefined `random` source-type, new measurements should provide random numbers as result.

Metric-Farmer provides the following predefined source types: `static`, `random` and `file_count`

Take a look into *Sources* to get details about them or to get information about how to define your own source types for your special needs.

CHAPTER 4

# Motivation

Metric-Farmer is based on the needs of a software development team inside a german automotive company.

The project team was searching for a small and practical way of measuring and analysing metrics related to the ISO 26262 standard for safety critical software.

Metric-Farmer is part of a software bundle, which was designed to support the development of ISO 26262 compliant software. Other tools are: sphinx-needs, sphinx-test-reports and tox-envreport.

## 4.1 Installation

You need a working Python installation with version 3.5 and above.

Please visit python.org , if Python needs to be installed.

### 4.1.1 Using pip

Using `pip` is the best way to install a package in Python.

Just open a terminal/command prompt and execute the following command:

```
pip install metricfarmer
```

### 4.1.2 Using sources

If you want to have the latest, not yet released version of Metric-Farmer, you must grab the source code and install it via setuptools. To do so, just follow the following instructions:

```
git clone https://github.com/useblocks/metricfarmer
python setup.py install
```

## 4.2 Cli

This pages describes the command line interface (cli) of Metric-Farmer and its options and arguments.



For a quick help, you can execute:

```
metricfarmer --help
```

As output you will get:

```
Usage: metricfarmer [OPTIONS] [TARGETS]...

  Measure metrics and execute TARGETS

Options:
  -f, --farmer TEXT   Only uses the given farm folder
  -m, --metrics TEXT  Filter metrics for specific name
  -t, --tags TEXT     Filter metrics for tags
  --list              Show configuration information only
  --help              Show this message and exit.
```

### 4.2.1 Targets argument

Set one or multiple `TARGETS` to get the wished result handling after measurement.

Example:

```
metricfarmer print file_json
```

You also can set no `TARGETS` argument, if you only want the measurement to be executed:

```
metricfarmer
```

### 4.2.2 Options

#### farmer

Use `-f` or `--farmer` to define a single folder, which contains your farm files.

All other locations will **not** be loaded.

So predefined farm-files from Metric-Farmer, from user/home directory and from the current working directory will be ignored.

Example:

```
metricfarmer --farmer /temp/farmer_folder/ print
```

#### metrics

Use `-f` or `--metrics` to define a comma separated list of metric names, which shall get measured. All other metrics will be ignored, if not also part of a *tag-filter*.

Example:

```
metricfarmer --metrics my_metric,another_metric print
```

#### tags

Use `-t` or `--tags` to define a comma separated list of tags, which metrics must have to get measured.

A metric needs only to match **one** tag of the complete list to be taken into account for measurement.

Can be combined with a *metric name filter*.

Example:

```
metricfarmer --tags mf_examples,my_tag print
```

#### list

Use `--list` to get detailed information of loaded extension and configurations files.

You also get lists of available metrics, tags, sources and targets.

Can **not** be combined with other options.

Example:

```
metricfarmer --list
```

#### help

Use `--help` to get the default help message.

Example:

```
metricfarmer --help
```

## 4.3 Metrics

Metrics can be defined inside any `.farm`-file and must contain a unique name and a source configuration, which describes how to measure the metric.

```
1  {
2    "metrics": {
3       "all_html_files": {
4          "source": {
5            "type": "html_file_count"
6          }
7       },
8       "doc_html_files": {
9          "source": {
10           "type": "html_file_count",
11           "path": "docs/"
12         }
13      }
14   }
15 }
```

Metrics must be defined in the **metrics** section, which stores a dictionary for specific metrics (line 2).

Each metric must be registered as element of this dictionary with an unique name (line 3 and 8). The name must be unique in all used `.farm`-files.

### 4.3.1 Parameters

A metric can contain multiple parameters, which can be simple numbers or strings or list and complex dictionaries.

#### description

A `description` can be set to simplify the understanding and maintenance of your metrics for other users.

```
{
  "metrics": {
     "my_metric": {
        "description": "An awesome metric to measure awesome stuff",
        "source": {
          "type": "static"
        }
     }
  }
}
```

#### source

`source` is the most important section of a metric, as it defines what gets measured and how.

```
1  {
2    "metrics": {
3       "doc_html_files": {
4          "source": {
5            "type": "html_file_count",
6            "path": "docs/"
7          }
8       }
9    }
10 }
```

`source` must be a dictionary and contain at least the parameter `type` (line 5).

`type` must be a string and reference an existing and loaded *source type*.

The used `source type` defines what other parameters can be set and are used during measurement. These parameters differs a lot between the different source types. So take a look into the related source type documentation.

However, there aren't any checks, if additional parameters are really supported by the referenced `source type`. So you are free to set as many parameters as you like.

## 4.4 Sources

A **source** defines and configures the way a metric shall get measured.

It normally contains a set of parameters, which are shared between all metrics, which use this source. But each metric can override specific parameters for its own measurement.

A metric must reference a `source type` defined in a `.farm`-file. And a `source type` needs to reference a `source class`, which is provided by Metric-Farmer extensions and defined as a function in a Python file.

**Page content**



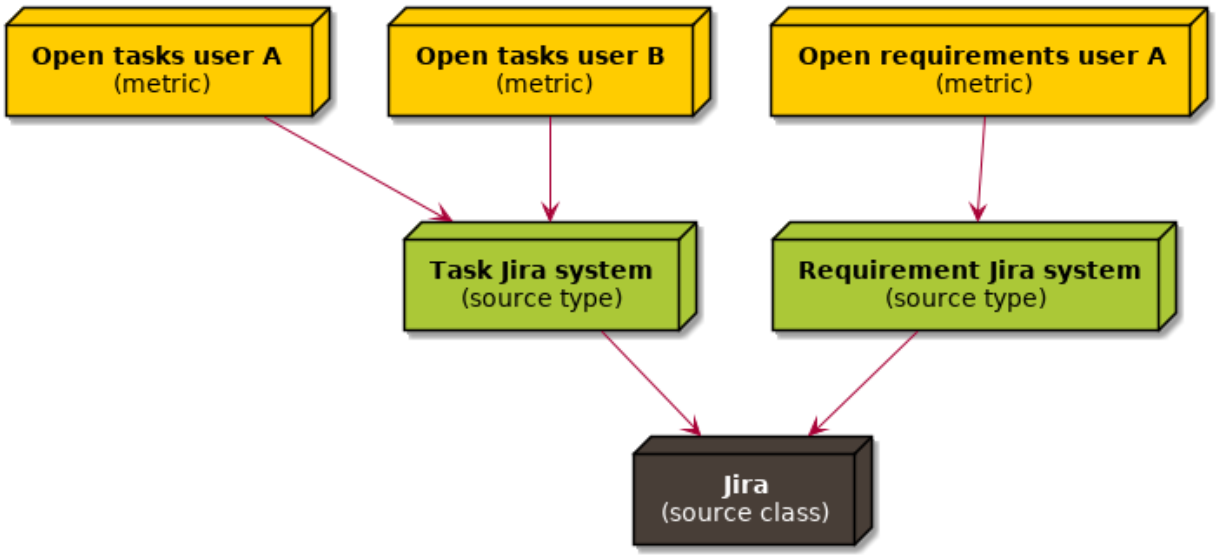
**Context example**

Image you are working for a company. This company is using **Jira** for their issue systems. Two different Jira installation exist: One for development tasks and one for product requirements.

The needed functions to access this Jira systems and measure metrics are the same, therefore only one `source class` is needed, which must be provided by an Metric-Farmer extension.

But both systems are available on different URLs and use different credentials. So two `source types` are needed, to store the different urls and credentials as parameters. These `source types` must be configured by the user in a `.farm`-file.

The needed metrics can then measure the needed JIRA system by referencing the correct `source type`. The metrics itself are only setting a filter parameter to get the needed data.




### 4.4.1 Source types

`Source types` are referenced by metrics and add a use-case specific configuration to a normally not configured `source class`.

**Example**: Referenced source in a metric definition

```
{
  "metrics": {
     "all_html_files": {
        "source": {
          "type": "html_file_count"
        }
     },
     "doc_html_files": {
        "source": {
          "type": "html_file_count",
          "path": "docs/"
        }
     }
  }
}
```

Both metrics use the same source `html_file_count`, but the last metric also overrides the `path` parameter.

**Example**: Referenced source class in a source definition

```
{
  "sources": {
    "html_file_count": {
      "class": "mf.file_count",
      "description": "Counts all html files in my project",
      "pattern": "**/*.html"
    }
  }
}
```

This source references the source class `mf.file_count`. `mf` is the namespace of the extension (here Metric-Farmer) and `file_count` is the source function to call.

### Predefined sources

Metric-Farmer provides the predefined sources `static`, `random` and `file_count`. They are used mainly for examples and simple use cases.

This is the content of the `.farm`-file, which defines the predefined sources:

```
{
  "sources": {
    "static": {
      "class": "mf.static",
      "description": "Sets a static value for a metric. Default is 0.",
      "value": 0
    },

    "random": {
      "class": "mf.random",
      "description": "Sets a random number between 0 and 10.",
      "min": 0,
      "max": 10,
      "digits": 2
    },

    "file_count": {
      "class": "mf.file_count",
      "description": "Counts all file in the current working directory."
    },

    "rest": {
      "class": "mf.rest",
      "description": "Makes a REST call against a given url."
    }

  }
}
```

**Usage example**:

```
{
  "metrics" : {
    "my_static_metric": {
      "type": "static",
      "value": 42
    },
    "my_random_metric": {
      "type": "random",
      "min": -50,
      "max": 50,
      "digits": 1
    },
    "my_file_count_metric": {
      "type": "file_count",
      "pattern": "**/*.farm"
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

#### Own sources

Sources get defined in the `sources` section of a `.farm`-file.

Example:

```
{
    "metrics": {}
    "sources": {
        "my_source": {
            "class": "mf.file_count",
            "description": "Counts all c-files in all subfolders"
            "pattern": "**/*.c"
        }
    }
    "targets": {}
}
```

They must have a `class` parameter, which must contain a string to reference a `source class` from a Metric-Farmer extension. Run `metricfarmer --list` to sell all available source classes.

They should also have a `description` parameter for documentation.

All other needed parameters are based on the selected `source class`. So please take a look into their documentation to find out which parameters are available and are mandatory.

Defined `sources` can be used and referenced in all other `.farm`-files.

### 4.4.2 Source classes

A `source class` is the link to a python function, which does the specific measurement work. They are provided by Metric-Farmer extensions.

For all available source classes on your installation, please execute `metricfarmer --list`.

#### Predefined source classes

Metric-Farmer provides already some basic source classes to support some frequent use cases.

You can use them inside your own `source type` definitions

Each `source class` has its own set of needed parameters. See their documentation for a detailed picture.

Available target classes are:

- *mf.static*
- *mf.random*
- *mf.file_count*
- *mf.rest*

### mf.static

`mf.static` allows to get static values from sources.

Table 1: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| value | A static value, which is used as result | None | No |

### mf.random

`mf.random` returns a random number as measurement result.

Use `min` and `max` parameters to define the range.

Table 2: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| min | Minimum value, which is allowed as result | 0.0 | No |
| max | Maximum value, which is allwoed as result | 100.0 | No |
| digits | Allowed positions after decimal point | 2 | No |
| sleep | Seconds to sleep/wait after measurement | 0 | No |

### mf.file_count

`mf.file_count` measure the amount of files inside a folder and its subfolders.

Use `pattern` to define what kind of files shall get measured.

Table 3: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| path | Path to a folder, where Metric-Farmer shall count the files | current working directory | No |
| pattern | Unix style pattern, which defines what to count. See docs for details. | `**/*` | No |

### mf.rest

`mf.rest` allows to take a specific part of the answer of a REST call as measurement result.

Table 4: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| url | Complete url for the request. | None | Yes |
| user | Username to use for Basic Authentication, if needed. | None | No |
| password | Password/Token to use for Basic Authentication, if needed. | None | No |
| headers | List of headers, which needs to be set for the call. | {'content-type': 'application/json'} | No |
| method | HTTP method type. Supported are GET, POST and PUT | GET | No |
| payload | Structured data, which will be send as json data. If method=GET, data must be a flat dictionary. | None | No |
| no_escape | If True, payload will be added to url unescaped (only GET). | False | No |
| result_call | Python statement, which gets evaluated and defines the location of the data, which shall be taken as measurement result | None | No |

`result_call` must be a Python based statement. This statement gets executed and has access to the request result object under the name `rest_result`.

Imagine the service sends back the following data:

```
{
   "name": "Frank",
   "age": 32,
   "friends": ["Peter", "Alex", "Sandra"]
}
```

If the `age` shall be taken as measurement result, the `result_call` must be `rest_result['age']`.

But even more complex measurements are possible. Lets say we need to measure the amount of friends. Then the `result_call` should be `len(rest_result['friends']))`.

**Example source type**

```
{
   "sources": {
      "company_service": {
         "class": "mf.rest",
         "url": "https://my-company.com/service/api/rest",
         "user": "my-name",
         "password": "my_password",
         "method": "POST",
         "payload": {
            "filter": "Issues = Open"
          },
         "result_call": "result['service_A']['total']"
      }
   }
}
```

### Own source classes

If you wish to create your own `source class` please take a look into *Extensions*.

## 4.5 Targets

Targets care about the final handling of metric results, after all measurements have been executed.

They are normally storing the results in a specific format on a specific system. E.g. locally as csv file or as entry on a metric system like Prometheus.

There is no direct link between a metric and a target, as a metric does not need to know, how its result get handled.

Targets get selected by the user during the call of Metric-Farmer, e.g. `metricfarmer print`, or by using the related *Settings*: *targets_default* or *targets_always*.

**Page content**



### 4.5.1 Target types

#### Predefined targets

Metric-Farmer provides the predefined targets `print`, `print_json`, `file_text`, `file_json`, `file_csv` and `db_sqlite`.

This is the content of the `.farm`-file, which defines the predefined sources:

```
{
  "targets": {
    "print": {
      "class": "mf.print",
      "description": "Prints the measurement results on std.out"
    },

    "print_json": {
      "class": "mf.print_json",
      "description": "Prints the measurement results as json on std.out"
    },

    "file_text": {
```

(continues on next page)

(continued from previous page)

```
      "class": "mf.file_text",
      "description": "Stores metric results in a simple text file.",
      "path": "metric_results.txt",
      "override": true
    },

    "file_json": {
      "class": "mf.file_json",
      "description": "Stores metric results in a json file.",
      "path": "metric_results.json",
      "override": true
    },

    "file_csv": {
      "class": "mf.file_csv",
      "description": "Stores metric results in a csv file.",
      "path": "metric_results.csv",
      "override": false,
      "delimiter": ","
    },

    "db_sqlite": {
      "class": "mf.db_sqlite",
      "description": "Stores metric results in a sqlite file.",
      "path": "metric_results.db",
      "table": "metrics"
    }
  }
}
```

Please take the above file content as reference about what defaults are set/used for each target. For instance the use file path of csv output.

The targets can be simply used as argument in the Metric-Farmer call: `metricfarmer db_sqlite`.

### Own targets

Targets can be defined in each `.farm`-file. After definition they can directly be selected and used by the user.

Example:

```
{
  "targets": {
    "file_csv": {
      "class": "mf.file_csv",
      "description": "Stores metric results in a csv file.",
      "path": "metric_results.csv",
      "override": false,
      "delimiter": ","
    }
  }
}
```

Targets must have an unique name, which is used only once in all loaded `.farm`-files.

They also need to reference a `target class` of a loaded extension:

```
"file_csv": {
  "class": "mf.file_csv"
}
```

The `target class` defines what other parameters are needed. So please take a look into the related documentation of target classes.

Run `metricfarmer --list` to get a list off all extension and their provided target classes.

### 4.5.2 Target classes

#### Predefined target classes

This list shows only target classes provided by Metric-Farmer. If you have installed some extensions, this list might be much bigger. Please run `metricfarmer --list` to see what is really available on your system.

#### mf.print

Prints the result on the command line.

Does not support any parameters.

#### mf.print_json

Prints the result as json format on the command line.

Does not support any parameters.

#### mf.file_text

Writes the results to a text file.

Table 5: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| path | File path to use | metric_results.txt | No |
| override | If true, existing file gets replace, otherwise an error is thrown | False | No |

#### mf.file_json

Writes the results to a json file.

Same out put as target `print_json`.

Table 6: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| path | File path to use | metric_results.txt | No |
| override | If true, existing file gets replace, otherwise an error is thrown | False | No |

### mf.file_csv

Writes the results to a csv file.

If `override` is set to false and a csv file already exists, new results with get added.

Table 7: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| path | File path to use | metric_results.txt | No |
| override | If true, existing file gets replace, otherwise an error is thrown | False | No |
| delimiter | Character to use as delimiter in csv file | , | No |

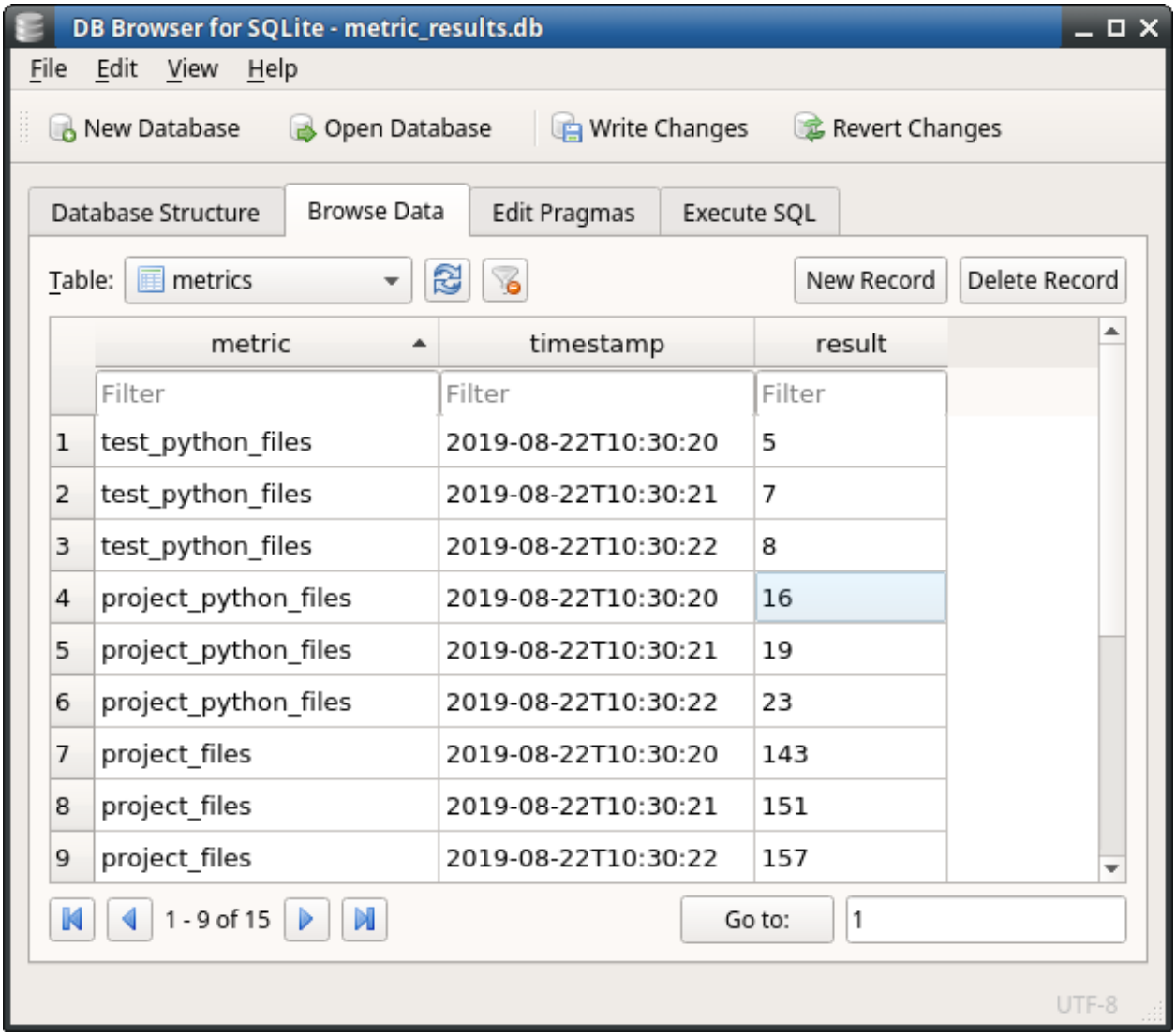
### mf.db_sqlite

Writes results to a local sqlite database file.

File path and table can be configured by the related parameters. But the needed table columns are fixed: `metric`, `timestamp`, `result`

New data is always added and existing data is kept.

The database table may look like this after several executions of `metricfarmer db_sqlite`.




You can use a sqlite database viewer like sqlitebrowser to analyze the data.

Table 8: Parameters

| Parameter | Description | Default | Required |
|---|---|---|---|
| path | File path to use | metric_results.db | No |
| table | Name of the table to use for string result data | metrics | No |

## 4.6 Settings

`Settings` are used to configure the Metric-Farmer Application.

They can be part of any `.farm`-file:

```
{
   "settings": {
      "setting_A": true,
      "setting_B": ["my_list"]
```

(continues on next page)

(continued from previous page)

```
        }
}
```

All available settings are defined by Metric-Farmer, as their usage is hardly coded in its source code.

Extensions have access to it, but should normally not use them. Instead their configuration should be stored in the parameters of their `sources` and `targets`.

### 4.6.1 Available settings



#### targets_default

List of default targets, which shall be used, if the user is using Metric-Farmer without any target.

Has no effect, if the user provides own targets in its call.

Example:

```
{
  "settings": {
    "targets_default": ["print"]
  }
}
```

Calling just `metricfarmer` will automatically execute the target `print`, if the above configuration is given.

#### targets_always

List of targets, which shall always get executed. No matter what the user has defined.

This targets will get executed first, then the user defined targets get executed.

Example:

```
{
  "settings": {
    "targets_always": ["print_json"]
  }
}
```

If the user runs `metricfarmer print` with the above configuration, the two targets `print_json` and `print` will get executed.

## 4.7 Extensions

Metric-Farmer can be easily extended by developers with own solutions for source and target types.

The reasons for this are the measurement from not yet supported sources (e.g. a department specific Excel list) or the storage of measurement results on not yet supported targets (e.g. a specific company metric system like Prometheus)

### 4.7.1 Concept

Metric-Farmer can be extend by using the entry-point mechanism of setuptools.

So a Metric-Farmer extension must be a valid Python package, which contains a `setup.py` file.

```
import os
from setuptools import setup

setup(
    name='Your extension',
      # ... More, but not for us important configurations
    entry_points={
        'metricfarmer': ['unimportant_name=your_package.your_modul:ExtensionClass']
    }
)
```

During installation of this package, the `entry_point` content gets registered on the used Python environment.

For Metric-Farmer the entry-point entry must be a class, which inherits from `metricfarmer.extensions.MetricFarmerExtension` and must have common variables, which define sources and target types.

Metric-Farmer creates an instance of this class during startup. From this point all defined sources and targets types of the extension are available.

A single Python package can register as many Metric-Farmer extensions as it likes.

### 4.7.2 Step by step introductions

1. Create a new folder `my_project`
2. Create a `setup.py` file and a `my_mf_extension.py` inside the above folder.
3. For `mf_my_extension.py` use the following content.

```
from metricfarmer.extensions import MetricFarmerExtension

def my_source_a(**kwargs):
   return 100

def my_target_a(metrics, **kwargs):
   for metric in metrics.keys():
      print metric

class MyExtension(MetricFarmerExtension):
    def __init__(self, app):
        self.app = app

        self.name = "My Extension"
        self.namespace = 'me'
        self.author = 'Awesome guy'
        self.description = 'Metric-Farmer extension...'

        self.source_classes = {
```

(continues on next page)

(continued from previous page)

```
            'my_source': my_source_a
        }

        self.target_classes = {
            'my_target': my_target_a
        }
```

4. Then register this class inside your `setup.py` file:

```
import os
from setuptools import setup, find_packages

setup(
    name='My extension project',
    version='0.0.1',
    license='MIT',
    author='Me',
    author_email='me@me.com',
    description='Collects and stores metrics for my project.',
    platforms='any',
    packages=find_packages(),
    install_requires=['metricfarmer'],
    entry_points={
        'metricfarmer': ['my_extension=my_project.mf_my_extension:MyExtension']
    }
)
```

5. After that you need to install your package, so that Python is aware of the new entry_point entry:

```
pip install -e .
```

6. Finally you should be able to address your source class with `me.my_source` and the target class with `me.my_target` in the related `class` parameters of source/target type definitions.

### 4.7.3 Example

Take a look into the source code of Metric Farmer, as it is using the entry-point mechanism to register all available sources and targets.

Visit https://github.com/useblocks/metricfarmer/tree/master/metricfarmer/extensions to get an overview about all folders and files.

The most important stuff is happening in file mf_extension.py. There the needed class gets defined.

This class is then used in the setup.py file as value for the `metricfarmer` entry-point.

## 4.8 Helpers

**Helpers** are little functions or definitions, which bring some helpful functions into the json-format of the `.farm`-files

All helpers are defined and configured as a string value and start with `:MF_` + helper name.

Current helpers are:

### 4.8.1 MF_REPLACE

MF_REPLACE replaces the value, where it is defined, with a value taken from another parameter. Short example: `:MF_REPLACE:my_name:` will search for `my_name` in the source definition of the current metric.

This helper is mainly used in source definitions, where single values need to be replaced by values from the metric definition.

**Example use case**

Imagine you need to perform a REST call on a webservice and the webservice needs a complex payload. Most parts of the payload are the same for all calls, but 1 option is specific for each metric.

In the below example we want to measure the amount of tickets in a ticket-system called Jira. Most important parameter is the jql-parameter (search string), which differs for each metric. But the rest of the needed configuration: data to get, amount of data to get, ... needs to be the same for each metric.

So instead a letting each metric definition contain the whole complex payload, it shall only define the **jql**. The complex payload is only defined once in the source definition and the nested jql-parameter gets replaced by the jql-data from the metric definition.

```
{
  "metrics": {
    "open_issues": {
      "description": "Measure open jira issues",
      "source": {
        "type": "jira",
        "jql": "status = Open"
      }
    },
    "closed_issues": {
      "description": "Measure closed jira issues",
      "source": {
        "type": "jira",
        "jql": "status in ('Closed','Done')"
      }
    },
  },

  "sources": {
    "jira": {
      "class": "mf.rest",
      "url": "https://my_jira.com",
      "payload": {
        "fields": [
          "status"
        ],
        "maxResults": 1,
        "jql": ":MF_REPLACE:jql"
      },
    }
  }
}
```

### 4.8.2 MF_ENV

MF_ENV replaces the value, where it is defined, with a value defined by a named environment variable.

Example: `:MF_ENV:my_password` will look for a environment variable called `my_password` and takes its content as value.

If the variable is not found, the string will not get replaced.

This mechanism is useful to prevent the storage of credentials inside `.farm`-files, which may be stored on public repositories.

```
{
  "metrics": {
    "open_issues": {
      "description": "Measure open jira issues",
      "source": {
        "type": "jira",
        "jql": "status = Open"
      }
    },
  },

  "sources": {
    "jira": {
      "class": "mf.rest",
      "url": "https://my_jira.com",
      "user": ":MF_ENV:JIRA_USER",
      "password": ":MF_ENV:JIRA_PASSWORD",
      "payload": {
        "fields": [
          "status"
        ],
        "maxResults": 1,
        "jql": ":MF_REPLACE:jql"
      },
    }
  }
}
```

## 4.9 Snippets

The following pages contain some information and mainly examples how to use and configure Metric-Farmer for specific use cases.

Feel free to contribute to this list by creating PRs on our github project.

### 4.9.1 Get metrics from GitHub

The below example gets the amount of issues from GitHub.

For the **GitHub api v3** please set `"no_escape": true` so that the filter string does not get escaped and is used as it is in the GET call against GitHub.

The used `filter` itself is using a not existing github account for the assignee field. So the result should always be **0**.

GitHub API documentation: https://developer.github.com/v3/search/

```
{
  "metrics": {
    "github_issues": {
      "source": {
        "type": "rest_github_v3",
        "filter": "repo:useblocks/metricfarmer+type:issue+state:open+assignee:invalid"
      }
    }
  },

  "sources": {
    "rest_github_v3": {
      "class": "mf.rest",
      "url": "https://api.github.com/search/issues",
       "method": "GET",
       "payload": {
         "q": ":MF_REPLACE:filter"
       },
      "no_escape": true,
       "result_call": "result['total_count']"

    }
  },

  "targets": {
    "print": {
      "class": "mf.print"
    }
  }
}
```

### 4.9.2 Get metrics from JIRA

The below example gets issues for an empty jql of a cloud-based JIRA system.

Empty jql string means that all issues are taken as result.

As no user/password are provided and this JIRA system as no free-accessible projects, the result should be **0**.

---

**Hint:** You may need to create a API-token for JIRA-cloud, because using your password in JIRA-cloud is not allowed. In this case, simply use the token for the **password** field.

See JIRA api token docs.

---

```
{
  "metrics": {
    "jira_issues": {
      "source": {
        "type": "rest_jira",
        "jql": ""
      }
    }
  },

  "sources": {
```

(continues on next page)

(continued from previous page)

```
    "rest_jira": {
      "class": "mf.rest",
      "url": "https://useblocks.atlassian.net/rest/api/3/search",
       "method": "POST",
       "payload": {
         "jql": ":MF_REPLACE:jql",
         "maxResults": 1,
         "fields": [ "summary"],
         "startAt": 0
       },
       "result_call": "result['total']"
    }
  },


  "targets": {
    "print": {
      "class": "mf.print"
    }
  }
}
```

### 4.9.3 Secure credentials

Nearly all web-services need valid credentials for authentication.

Instead of storing this security related information inside `.farm` -files, you should store them on local Environment variables , which are normally not available for other users.

Use the helper function *MF_ENV* to get this data into your configuration during runtime of Metric-Farmer.

```
{
  "metrics": {
    "open_issues": {
      "description": "Measure open jira issues",
      "source": {
        "type": "jira",
        "jql": "status = Open"
      }
    },
  },

  "sources": {
    "jira": {
      "class": "mf.rest",
      "url": "https://my_jira.com",
      "user": ":MF_ENV:JIRA_USER",
      "password": ":MF_ENV:JIRA_PASSWORD",
      "payload": {
        "fields": [
          "status"
        ],
        "maxResults": 1,
        "jql": ":MF_REPLACE:jql"
      },
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

## 4.10 Changelog

### 4.10.1 0.2.0

- Starting changelog
- Diffs to 0.0.1: helper function, rest support, better documentation.