reconplogger Documentation

Release 4.8.1

Mauricio Villegas

Apr 12, 2021

CONTENTS

This repository contains the code of reconplogger, a python package intended to ease the standardization of logging within omni:us. The main design decision of reconplogger is to allow total freedom to reconfigure loggers without hard coding anything.

The package contains essentially three things:

- A default logging configuration.
- A function for loading logging configuration for regular python code.
- A function for loading logging configuration for flask-based microservices.
- An inheritable class to add a logger property.
- Lower level functions for:
  - Loading logging configuration from any of: config file, environment variable, or default.
  - Replacing the handlers of an existing Logger object.
  - Function to add a file handler to a logger.

CHAPTER

ONE

HOW TO USE

There are two main use cases reconplogger targets. One is for logging in regular generic python code and the second one is logging in microservices. See the two standardizing sections below for a detailed explanation of the two use cases.

1.1 Add as requirement

The first step to use reconplogger is adding it as a requirement in the respective package where it will be used. This means adding it in the file *setup.cfg* as an item in `install_requires` or in an `extras_require` depending on whether reconplogger is intended to be a core or an optional requirement.

Note: It is highly discouraged to develop packages in which requirements are added directly to *setup.py* or to have an ambiguous *requirements.txt* file. See the *setup.cfg* file in the reconplogger source code for reference.

1.2 Default logging configuration

A feature that reconplogger provides is the possibility of externally setting the logging configuration without having to change code or implement any parsing of configuration. However, if a logging configuration is not given externally, reconplogger provides a default configuration.

The default configuration defines three handlers, two of which are stream handlers and are set to DEBUG log level. The first handler called `plain_handler` uses a simple plain text formatter, and the second handler called `json_handler` as the name suggests outputs in json format, using the logmatic JsonFormatter class. The third handler called `null_handler` is useful to disable all logging.

For each handler the default configuration defines a corresponding logger: `plain_logger`, `json_logger` and `null_logger`.

1.3 Standardizing logging in regular python

One objective of reconplogger is to ease the use of logging and standardize the way it is done across all omni:us python code. The use of reconplogger comes down to calling one function to get the logger object. For regular python code (i.e. not a microservice) the function to use is *`reconplogger.logger_setup()`*.

The following code snippet illustrates the use:

```
import reconplogger

# Default plain logger
logger = reconplogger.logger_setup()
logger.info('My log message')

# Json logger and custom prefix
logger = reconplogger.logger_setup('json_logger', env_prefix='MYAPP')
logger.info('My log message in json format')
```

This function gives you the ability to set the default logger to use (`logger_name` argument whose default value is `plain_logger`) and optionally provide a logging `config` and/or a logging `level` that overrides the level in the config.

All of these values can be overridden via environment variables whose names are prefixed by the value of the `env_prefix` argument. The environment variables supported are: `{env_prefix}_CFG`, `{env_prefix}_NAME` and `{env_prefix}_LEVEL`. Note that the environment variable names are not required to be prefixed by the default `env_prefix='LOGGER'`. The prefix can be chosen by the user for each particular application.

For functions or classes that receive logger object as an argument, it might be desired to set a non-logging default so that it can be called without specifying one. For this reconplogger defines `null logger` that could be used as follows:

```
from reconplogger import null_logger

...

def my_func(arg1, arg2, logger=null_logger):

...
```

1.4 Standardizing logging in flask-based microservices

The most important objective of reconplogger is to allow standardization of a structured logging format for all microservices developed. Thus, the logging from all microservices should be configured like explained here. The use is analogous to the previous case, but using the function *`reconplogger.flask_app_logger_setup()`* instead, and giving as first argument the flask app object.

Additional to the previous case, this function:

- Replaces the flask app and werkzeug loggers to use a reconplogger configured one.
- Add to the logs the correlation_id
- Add before and after request functions to log the request details when the request is processed
- Patch the *requests* library forwarding the correlation id in any call to other microservices

What is the correlation ID? In a system build with microservices we need a way to correlate logs coming from different microservices to the same "external" call. For example when a user of our system do a call to the MicroserviceA this could need to retrieve some information from the MicroserviceB, if there is an error and we want to check the logs of the MicroserviceB related to the user call we don't have a way to correlate them, to solve this we use the correlation id! Its a uuid4 that its passed in the headers of the rest calls and will be forwarded automatically when we do calls with the library *requests*, if the correlation id its not present in the request headers it will be generated, all of this is taken care in the background by this library.

The usage would be as follows:

```
import reconplogger
from flask import Flask

...

app = Flask(__name__)

...

logger = reconplogger.flask_app_logger_setup(app, level='DEBUG')

## NOTE: do not change logger beyond this point!

...

## Use logger in code
myclass = MyClass(..., logger=logger)

...

## User logger in a flask request
@app.route('/')
def hello_world():
    logger.info('i like logs')
    correlation_id = reconplogger.get_correlation_id()
    logger.info('correlation id for this request: '+correlation_id)
    return 'Hello, World!'

...
```

As illustrated in the previous example the `get_correlation_id()` function can be used to get the correlation id for the current application context. However, there are cases in which it is desired to set the correlation id, instead of getting a randomly generated one. In this case the `get_correlation_id()` function is used, for example as follows:

```
@app.route('/')
def hello_world():
    reconplogger.set_correlation_id('my_correlation_id')
    logger.info('i like logs')
    return 'Hello, World!'
```

An important note is that after configuring the logger, the code should not modify the logger configuration. For example, the logging level should not be modified. Adding an additional handler to the logger is not a problem. This could be desired for example to additionally log to a file.

In the helm *values.yaml* file of the microservice, the default values for the environment variables should be set as:

```
LOGGER_CFG:
LOGGER_NAME: json_logger
LOGGER_LEVEL: DEBUG
```

With the `json_logger` logger, the format of the logs should look something like the following:

```
{"asctime": "2018-09-05 17:38:38,137", "levelname": "INFO", "filename": "test_
↪formatter.py", "lineno": 5, "message": "Hello world"}
{"asctime": "2018-09-05 17:38:38,137", "levelname": "DEBUG", "filename": "test_
↪formatter.py", "lineno": 9, "message": "Hello world"}
```

(continues on next page)

(continued from previous page)

```
{"asctime": "2018-09-05 17:38:38,137", "levelname": "ERROR", "filename": "test_
↪formatter.py", "lineno": 13, "message": "Hello world"}
{"asctime": "2018-09-05 17:38:38,137", "levelname": "CRITICAL", "filename": "test_
↪formatter.py", "lineno": 17, "message": "Hello world"}
{"asctime": "2018-09-05 17:38:38,137", "levelname": "ERROR", "filename": "test_
↪formatter.py", "lineno": 25, "message": "division by zero"}
{"asctime": "2018-09-05 17:38:38,138", "levelname": "ERROR", "filename": "test_
↪formatter.py", "lineno": 33, "message": "Exception has occured", "exc_info":
↪"Traceback (most recent call last):\n  File \"reconplogger/tests/test_formatter.py\
↪", line 31, in test_exception_with_trace\n    b = 100 / 0\nZeroDivisionError:␣
↪division by zero"}
{"asctime": "2018-09-05 17:38:38,138", "levelname": "INFO", "filename": "test_
↪formatter.py", "lineno": 37, "message": "Hello world", "context check": "check"}

{"asctime": "2020-09-02 17:20:16,428", "levelname": "INFO", "filename": "hello.py",
↪"lineno": 12, "message": "i like logs", "correlation_id": "3958f378-5d48-4e1c-b83b-
↪3c6d9f95faec"}
{"asctime": "2020-09-02 17:20:16,428", "levelname": "INFO", "filename": "reconplogger.
↪py", "lineno": 271, "message": "Request is completed", "http_endpoint": "/", "http_
↪method": "GET", "http_response_code": 200, "http_response_size": 56, "http_input_
↪payload_size": null, "http_input_payload_type": null, "http_response_time": "0.
↪0002014636993408203", "correlation_id": "3958f378-5d48-4e1c-b83b-3c6d9f95faec"}
```

1.5 Use of the logger object

The logger objects returned by the setup functions are normal python `logging.Logger` objects, so all the standard logging functionalities should be used. Please refer to the logging package documentation for details.

A couple of logging features that should be very commonly used are the following. To add additional structured information to a log, the `extra` argument should be used. A simple example could be:

```
logger.info('Successfully processed document', extra={'uuid': uuid})
```

When an exception occurs the `exc_info=True` argument should be used, for example:

```
try:
    ...
except:
    logger.critical('Failed to run task', exc_info=True)
```

1.6 Adding a file handler

In some circumstances it is desired to add to a logger a file handler so that the logging messages are also saved to a file. This normally requires at least three lines of code, thus to simplify things reconplogger provides the *`reconplogger.add_file_handler()`* function to do the same with a single line of code. The use is quite straightforward as:

```
reconplogger.add_file_handler(logger, '/path/to/log/file.log')
```

1.7 Adding a logging property

When implementing classes it is common to add logging support to it. For this an inheritable class *RLoggerProperty* is included in reconplogger to add an `rlogger` property to easily set and get the reconplogger logger. An example of use is the following:

```
from reconplogger import RLoggerProperty

class MyClass(RLoggerProperty):
    def __init__(self, logger):
        self.rlogger = logger
    def my_method(self):
        self.rlogger.error('my_method was called')

MyClass(logger=True).my_method()
```

1.8 Overriding logging configuration

An important feature of reconplogger is that the logging configuration of apps that use it can be easily changed via the environment variables. First set the environment variables with the desired logging configuration and logger name:

```
export LOGGER_NAME="example_logger"

export LOGGER_CFG='{
    "version": 1,
    "formatters": {
        "verbose": {
            "format": "%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d
↪%(message)s"
        }
    },
    "handlers": {
        "console":{
            "level":"DEBUG",
            "class":"logging.StreamHandler",
            "formatter": "verbose"
        }
    },
    "loggers": {
        "example_logger": {
            "handlers": ["console"],
            "level": "ERROR",
        }
    }
}'
```

Then, in the python code the logger would be used as follows:

```
>>> import reconplogger
>>> logger = reconplogger.logger_setup(env_prefix='LOGGER')
>>> logger.error('My error message')
ERROR 2019-10-18 14:45:22,629 <stdin> 16876 139918773925696 My error message
```

CHAPTER

TWO

LOW LEVEL FUNCTIONS

2.1 Loading configuration

The *reconplogger.load_config()* function allows loading of a python logging configuration. The format config can be either json or yaml. The loading of configuration can be from a file (giving its path), from an environment variable (giving the variable name), a raw configuration string, or loading the default configuration that comes with reconplogger. See below examples of loading for each of the cases:

```
import reconplogger

## Load from config file
reconplogger.load_config('/path/to/config.yaml')

## Load from environment variable
reconplogger.load_config('LOGGER_CFG')

## Load default config
reconplogger.load_config('reconplogger_default_cfg')
```

2.2 Replacing logger handlers

In some cases it might be needed to replace the handlers of some already existing logger. For this reconplogger provides the *reconplogger.replace_logger_handlers()* function. To use it, simply provide the logger in which to replace the handlers and the logger from where to get the handlers. The procedure would be as follows:

```
import reconplogger

logger = reconplogger.logger_setup('json_logger')
reconplogger.replace_logger_handlers('some_logger_name', logger)
```

CHAPTER

THREE

CONTRIBUTING

Contributions to this package are very welcome. When you plan to work with the source code, note that this project does not include a *requirements.txt* file. This is by intention. To make it very clear what are the requirements for different use cases, all the requirements of the project are stored in the file *setup.cfg*. The basic runtime requirements are defined in section `[options]` in the `install_requires` entry. All optional requirements are stored in section `[options.extras_require]`. There are `test`, `dev` and `doc` extras require to be used by developers (e.g. requirements to run the unit tests) and an `all` extras require for optional runtime requirements, namely Flask support.

The recommended way to work with the source code is the following. First clone the repository, then create a virtual environment, activate it and finally install the development requirements. More precisely the steps would be:

```
git clone https://github.com/omni-us/reconplogger.git
cd reconplogger
virtualenv -p python3 venv
. venv/bin/activate
```

The crucial step is installing the requirements which would be done by running:

```
pip3 install --editable ".[dev]"
```

Running the unit tests can be done either using using tox or the `setup.py` script. The unit tests are also installed with the package, thus can be used to in a production system.

```
tox  # Run tests using tox
./setup.py test_coverage  # Run tests and generate coverage report
python3 -m reconplogger_tests  # Run tests for installed package
```

3.1 Pull requests

- To contribute it is required to create and push to a new branch and issue a pull request.
- A pull request will only be accepted if:
  - All python files pass pylint checks.
  - All unit tests run successfully.
  - New code has docstrings and gets included in the html documentation.
- When developing, after cloning be sure to run the githook-pre-commit to setup the pre-commit hook. This will help you by automatically running pylint before every commit.

3.2 Using bump version

Only the maintainer of this repo should bump versions and this should be done only on the master branch. To bump the version it is required to use the bumpversion command that should already be installed if `pip3 install --editable .[dev,doc,test,all]` was run as previously instructed.

```
bumpversion major/minor/patch
```

Push the tags to the repository as well.

```
git push; git push --tags
```

When the version tags are pushed, circleci will automatically build the wheel file, test it and if successful, push the package to pypi.

CHAPTER

FOUR

DOCUMENTATION CONTENTS

4.1 API Reference

4.1.1 reconplogger module

Classes:

RLoggerProperty(*args, **kwargs)	Class designed to be inherited by other classes to add an rlogger property.

Functions:

add_file_handler(logger, file_path[, ...])	Adds a file handler to a given logger.
flask_app_logger_setup(flask_app[, ...])	Sets up logging configuration, configures flask to use it, and returns the logger.
get_correlation_id()	Returns the current correlation id.
get_logger(logger_name)	Returns an already existing logger.
load_config([cfg])	Loads a logging configuration from path or environment variable or dictionary object.
logger_setup([logger_name, config, level, ...])	Sets up logging configuration and returns the logger.
replace_logger_handlers(logger, handlers)	Replaces the handlers of a given logger.
set_correlation_id(correlation_id)	Sets the correlation id for the current application context.
test_logger(logger)	Logs one message to each debug, info and warning levels intended for testing.

class reconplogger.**RLoggerProperty**(**args*, ***kwargs*)

Bases: object

Class designed to be inherited by other classes to add an rlogger property.

Attributes:

rlogger	The logger property for the class.

property rlogger

The logger property for the class.

Getter Returns the current logger.

Setter Sets the reconplogger logger if True or sets null_logger if False or sets the given logger.

Raises **ValueError** – If an invalid logger value is given.

reconplogger.**add_file_handler**(*logger*, *file_path*, *format='%(asctime)s\\t%(levelname)s -- %(filename)s:%(lineno)s -- %(message)s'*, *level='DEBUG'*)

Adds a file handler to a given logger.

Parameters

- **logger** (Logger) – Logger object where to add the file handler.
- **file_path** (str) – Path to log file for handler.
- **format** (str) – Format for logging.
- **level** (Optional[str]) – Logging level for the handler.

reconplogger.**flask_app_logger_setup**(*flask_app*, *logger_name='plain_logger'*, *config=None*, *level=None*, *env_prefix='LOGGER'*, *parent=None*)

Sets up logging configuration, configures flask to use it, and returns the logger.

Parameters

- **flask_app** (*flask.app.Flask*) – The flask app object.
- **logger_name** (str) – Name of the logger that needs to be used.
- **config** (Optional[str]) – Configuration string or path to configuration file or configuration file via environment variable.
- **level** (Optional[str]) – Optional logging level that overrides one in config.
- **env_prefix** (str) – Environment variable names prefix for overriding logger configuration.
- **parent** (Optional[Logger]) – Set for logging delegation to the parent.

Return type Logger

Returns The logger object.

reconplogger.**get_correlation_id**()

Returns the current correlation id.

Raises

- **ImportError** – When flask package not available.
- **RuntimeError** – When run outside an application context or if flask app has not been setup.

Return type str

reconplogger.**get_logger**(*logger_name*)

Returns an already existing logger.

Parameters **logger_name** (str) – Name of the logger to get.

Return type Logger

Returns The logger object.

Raises **ValueError** – If the logger does not exist.

reconplogger.**load_config**(*cfg=None*)

Loads a logging configuration from path or environment variable or dictionary object.

Parameters **cfg** (Union[str, dict, None]) – Path to configuration file (json|yaml), or name of environment variable (json|yaml) or configuration object or None/"reconplogger_default_cfg" to use default configuration.

Returns The logging package object.

reconplogger.**logger_setup**(*logger_name='plain_logger'*, *config=None*, *level=None*, *env_prefix='LOGGER'*, *parent=None*, *init_messages=False*)

Sets up logging configuration and returns the logger.

Parameters

- **logger_name** (str) – Name of the logger that needs to be used.
- **config** (Optional[str]) – Configuration string or path to configuration file or configuration file via environment variable.
- **level** (Optional[str]) – Optional logging level that overrides one in config.
- **env_prefix** (str) – Environment variable names prefix for overriding logger configuration.
- **parent** (Optional[Logger]) – Set for logging delegation to the parent.
- **init_messages** (bool) – Whether to log init and test messages.

Return type Logger

Returns The logger object.

reconplogger.**replace_logger_handlers**(*logger*, *handlers*)

Replaces the handlers of a given logger.

Parameters

- **logger** (Union[Logger, str]) – Object or name of logger to replace handlers.
- **handlers** (Union[Logger, str]) – Object or name of logger from which to get handlers.

reconplogger.**set_correlation_id**(*correlation_id*)

Sets the correlation id for the current application context.

Raises

- **ImportError** – When flask package not available.
- **RuntimeError** – When run outside an application context or if flask app has not been setup.

reconplogger.**test_logger**(*logger*)

Logs one message to each debug, info and warning levels intended for testing.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX