structlog Documentation

Release

Author

November 07, 2015

Contents

Release v0.4.2 (What's new?).

structlog makes structured logging in Python easy by *augmenting* your *existing* logger. It allows you to split your log entries up into key/value pairs and build them incrementally without annoying boilerplate code.

It's licensed under Apache License, version 2, available from PyPI, the source code can be found on GitHub, the documentation at http://www.structlog.org/.

structlog targets Python 2.6, 2.7, 3.2, 3.3, and PyPy with no additional dependencies for core functionality.

If you need any help, visit us on `#structlog` on Freenode!

CHAPTER 1

The Pitch

structlog makes structured logging with *incremental context building* and *arbitrary formatting* as easy as:

```
>>> from structlog import get_logger
>>> log = get_logger()
>>> log = log.bind(user='anonymous', some_key=23)
>>> log = log.bind(user='hynek', another_key=42)
>>> log.info('user.logged_in', happy=True)
some_key=23 user='hynek' another_key=42 happy=True event='user.logged_in'
```

Please note that this example does *not* use standard library logging (but could so *easily*). The logger that's returned by `get_logger()` is *freely* configurable and uses a simple `PrintLogger` by default.

For…

- …reasons why structured logging in general and structlog in particular are the way to go, consult Why….
- …more realistic examples, peek into Examples.
- …getting started right away, jump straight into Getting Started.

Since structlog avoids monkey-patching and events are fully free-form, you can start using it **today**!

CHAPTER 2

User's Guide

2.1 Basics

2.1.1 Why...

...Structured Logging?

> I believe the widespread use of format strings in logging is based on two presumptions:
>
> - The first level consumer of a log message is a human.
> - The programmer knows what information is needed to debug an issue.
>
> I believe these presumptions are **no longer correct** in server side software.
>
> —Paul Querna

Structured logging means that you don't write hard-to-parse and hard-to-keep-consistent prose in your logs but that you log *events* that happen in a *context* instead.

...structlog?

Because it's easy and you don't have to replace your underlying logger – you just add structure to your log entries and format them to strings before they hit your real loggers.

structlog supports you with building your context as you go (e.g. if a user logs in, you bind their user name to your current logger) and log events when they happen (i.e. the user does something log-worthy):

```
>>> from structlog import get_logger
>>> log = get_logger()
>>> log = log.bind(user='anonymous', some_key=23)
>>> log = log.bind(user='hynek', another_key=42)
>>> log.info('user.logged_in', happy=True)
some_key=23 user='hynek' another_key=42 happy=True event='user.logged_in'
```

This ability to bind key/values pairs to a logger frees you from using conditionals, closures, or boilerplate methods to log out all relevant data.

Additionally, structlog offers you a flexible way to *filter* and *modify* your log entries using so called *processors* before the entry is passed to your real logger. The possibilities include *logging in JSON*, adding arbitrary meta data like *timestamps*, counting events as metrics, or *dropping log entries* caused by your monitoring system. structlog is also flexible enough to allow transparent *thread local* storage for your context if you don't like the idea of local bindings as in the example above.

2.1.2 Getting Started

Installation

structlog can be easily installed using:

```
$ pip install structlog
```

Python 2.6

If you're running Python 2.6 and want to use `OrderedDicts` for your context (which is the default), you also have to install the respective compatibility package:

```
$ pip install ordereddict
```

If the order of the keys of your context doesn't matter (e.g. if you're logging JSON that gets parsed anyway), simply use a vanilla `dict` to avoid this dependency. See Configuration on how to achieve that.

Your First Log Entry

A lot of effort went into making structlog accessible without reading pages of documentation. And indeed, the simplest possible usage looks like this:

```
>>> import structlog
>>> log = structlog.get_logger()
>>> log.msg('greeted', whom='world', more_than_a_string=[1, 2, 3])
whom='world' more_than_a_string=[1, 2, 3] event='greeted'
```

Here, structlog takes full advantage of its hopefully useful default settings:

- Output is sent to standard out instead of exploding into the user's face. Yes, that seems a rather controversial attitude towards logging.
- All keywords are formatted using *`structlog.processors.KeyValueRenderer`*. That in turn uses repr() to serialize all values to strings. Thus, it's easy to add support for logging of your own objects[1].

It should be noted that even in most complex logging setups the example would still look just like that thanks to *Configuration*.

There you go, structured logging! However, this alone wouldn't warrant its own package. After all, there's even a recipe on structured logging for the standard library. So let's go a step further.

Building a Context

Imagine a hypothetical web application that wants to log out all relevant data with just the API from above:

```
from structlog import get_logger

log = get_logger()


def view(request):
    user_agent = request.get('HTTP_USER_AGENT', 'UNKNOWN')
```

[1] In production, you're more likely to use *`JSONRenderer`* that can also be customized using a `__structlog__` method so you don't have to change your repr methods to something they weren't originally intended for.

```
    peer_ip = request.client_addr
    if something:
        log.msg('something', user_agent=user_agent, peer_ip=peer_ip)
        return 'something'
    elif something_else:
        log.msg('something_else', user_agent=user_agent, peer_ip=peer_ip)
        return 'something_else'
    else:
        log.msg('else', user_agent=user_agent, peer_ip=peer_ip)
        return 'else'
```

The calls themselves are nice and straight to the point, however you're repeating yourself all over the place. At this point, you'll be tempted to write a closure like

```
def log_closure(event):
   log.msg(event, user_agent=user_agent, peer_ip=peer_ip)
```

inside of the view. Problem solved? Not quite. What if the parameters are introduced step by step? Do you really want to have a logging closure in each of your views?

Let's have a look at a better approach:

```
from structlog import get_logger

logger = get_logger()


def view(request):
    log = logger.bind(
        user_agent=request.get('HTTP_USER_AGENT', 'UNKNOWN'),
        peer_ip=request.client_addr,
    )
    foo = request.get('foo')
    if foo:
        log = log.bind(foo=foo)
    if something:
        log.msg('something')
        return 'something'
    elif something_else:
        log.msg('something_else')
        return 'something_else'
    else:
        log.msg('else')
        return 'else'
```

Suddenly your logger becomes your closure!

For structlog, a log entry is just a dictionary called *event dict[ionary]*:

- You can pre-build a part of the dictionary step by step. These pre-saved values are called the *context*.
- As soon as an *event* happens – which is a dictionary too – it is merged together with the *context* to an *event dict* and logged out.
- To keep as much order of the keys as possible, an OrderedDict is used for the context by default.
- The recommended way of binding values is the one in these examples: creating new loggers with a new context. If you're okay with giving up immutable local state for convenience, you can also use *thread/greenlet local storage* for the context.

structlog and Standard Library's logging

structlog's primary application isn't printing though. Instead, it's intended to wrap your *existing* loggers and **add** *structure* and *incremental context building* to them. For that, structlog is *completely* agnostic of your underlying logger – you can use it with any logger you like.

The most prominent example of such an 'existing logger' is without doubt the logging module in the standard library. To make this common case as simple as possible, structlog comes with some tools to help you:

```
>>> import logging
>>> logging.basicConfig()
>>> from structlog import get_logger, configure
>>> from structlog.stdlib import LoggerFactory
>>> configure(logger_factory=LoggerFactory())
>>> log = get_logger()
>>> log.warn('it works!', difficulty='easy')
WARNING:structlog...:difficulty='easy' event='it works!'
```

In other words, you tell structlog that you would like to use the standard library logger factory and keep calling *get_logger()* like before.

Since structlog is mainly used together with standard library's logging, there's more goodness to make it as fast and convenient as possible.

Liked what you saw?

Now you're all set for the rest of the user's guide. If you want to see more code, make sure to check out the *Examples*!

2.1.3 Loggers

Bound Loggers

The center of structlog is the immutable log wrapper *BoundLogger*.

All it does is:

- Keep a *context dictionary* and a *logger* that it's wrapping,
- recreate itself with (optional) *additional* context data (the *bind()* and *new()* methods),
- recreate itself with *less* data (*unbind()*),
- and finally relay *all* other method calls to the wrapped logger[2] after processing the log entry with the configured chain of *processors*.

You won't be instantiating it yourself though. For that there is the *structlog.wrap_logger()* function (or the convenience function *structlog.get_logger()* we'll discuss in a minute):

```
>>> from structlog import wrap_logger
>>> class PrintLogger(object):
...     def msg(self, message):
...         print message
>>> def proc(logger, method_name, event_dict):
...     print 'I got called with', event_dict
...     return repr(event_dict)
>>> log = wrap_logger(PrintLogger(), processors=[proc], context_class=dict)
```

[2] Since this is slightly magicy, structlog comes with concrete loggers for the Python Standard Library and Twisted that offer you explicit APIs for the supported logging methods but behave identically like the generic BoundLogger otherwise.

```
>>> log2 = log.bind(x=42)
>>> log == log2
False
>>> log.msg('hello world')
I got called with {'event': 'hello world'}
{'event': 'hello world'}
>>> log2.msg('hello world')
I got called with {'x': 42, 'event': 'hello world'}
{'x': 42, 'event': 'hello world'}
>>> log3 = log2.unbind('x')
>>> log == log3
True
>>> log3.msg('nothing bound anymore', foo='but you can structure the event too')
I got called with {'foo': 'but you can structure the event too', 'event': 'nothing bound anymore'}
{'foo': 'but you can structure the event too', 'event': 'nothing bound anymore'}
```

As you can see, it accepts one mandatory and a few optional arguments:

logger The one and only positional argument is the logger that you want to wrap and to which the log entries will be proxied. If you wish to use a *configured logger factory*, set it to *None*.

processors A list of callables that can *filter, mutate, and format* the log entry before it gets passed to the wrapped logger.

Default is `[format_exc_info(), KeyValueRenderer]`.

context_class The class to save your context in. Particularly useful for *thread local context storage*.

Default is OrderedDict.

Additionally, the following arguments are allowed too:

wrapper_class A class to use instead of `BoundLogger` for wrapping. This is useful if you want to sub-class BoundLogger and add custom logging methods. BoundLogger's bind/new methods are sub-classing friendly so you won't have to re-implement them. Please refer to the *related example* for how this may look.

initial_values The values that new wrapped loggers are automatically constructed with. Useful for example if you want to have the module name as part of the context.

Note: Free your mind from the preconception that log entries have to be serialized to strings eventually. All structlog cares about is a *dictionary* of *keys* and *values*. What happens to it depends on the logger you wrap and your processors alone.

This gives you the power to log directly to databases, log aggregation servers, web services, and whatnot.

Printing and Testing

To save you the hassle of using standard library logging for simple standard out logging, structlog ships a `PrintLogger` that can log into arbitrary files – including standard out (which is the default if no file is passed into the constructor):

```
>>> from structlog import PrintLogger
>>> PrintLogger().info('hello world!')
hello world!
```

It's handy for both examples and in combination with tools like runit or stdout/stderr-forwarding.

Additionally – mostly for unit testing – structlog also ships with a logger that just returns whatever it gets passed into it: `ReturnLogger`.

```
>>> from structlog import ReturnLogger
>>> ReturnLogger().msg(42) == 42
True
>>> obj = ['hi']
>>> ReturnLogger().msg(obj) is obj
True
>>> ReturnLogger().msg('hello', when='again')
(('hello',), {'when': 'again'})
```

2.1.4 Configuration

Global Defaults

To make logging as unintrusive and straight-forward to use as possible, structlog comes with a plethora of configuration options and convenience functions. Let me start at the end and introduce you to the ultimate convenience function that relies purely on configuration: *structlog.get_logger()* (and its Twisted-friendly alias *structlog.getLogger()*).

The goal is to reduce your per-file logging boilerplate to:

```
from structlog.stdlib import get_logger
logger = get_logger()
```

while still giving you the full power via configuration.

To achieve that you'll have to call *structlog.configure()* on app initialization (of course, only if you're not content with the defaults). The *example* from the previous chapter could thus have been written as following:

```
>>> configure(processors=[proc], context_class=dict)
>>> log = wrap_logger(PrintLogger())
>>> log.msg('hello world')
I got called with {'event': 'hello world'}
{'event': 'hello world'}
```

In fact, it could even be written like

```
>>> configure(processors=[proc], context_class=dict)
>>> log = get_logger()
>>> log.msg('hello world')
I got called with {'event': 'hello world'}
{'event': 'hello world'}
```

because `PrintLogger` is the default LoggerFactory used (see *Logger Factories*).

structlog tries to behave in the least surprising way when it comes to handling defaults and configuration:

1. Arguments passed to *structlog.wrap_logger()* *always* take the highest precedence over configuration. That means that you can overwrite whatever you've configured for each logger respectively.
2. If you leave them on *None*, structlog will check whether you've configured default values using *structlog.configure()* and uses them if so.
3. If you haven't configured or passed anything at all, the default fallback values are used which means OrderedDict for context and `[`*StackInfoRenderer*, *format_exc_info()*, *KeyValueRenderer*`]` for the processor chain, and *False* for *cache_logger_on_first_use*.

If necessary, you can always reset your global configuration back to default values using *structlog.reset_defaults()*. That can be handy in tests.

Note: Since you will call *structlog.wrap_logger()* (or one of the `get_logger()` functions) most likely at import time and thus before you had a chance to configure structlog, they return a **proxy** that returns a correct wrapped logger on first `bind()`/`new()`.

Therefore, you must not call `new()` or `bind()` in module scope! Use *get_logger()*'s `initial_values` to achieve pre-populated contexts.

To enable you to log with the module-global logger, it will create a temporary BoundLogger and relay the log calls to it on *each call*. Therefore if you have nothing to bind but intend to do lots of log calls in a function, it makes sense performance-wise to create a local logger by calling `bind()` or `new()` without any parameters. See also Performance.

Logger Factories

To make *structlog.get_logger()* work, one needs one more option that hasn't been discussed yet: `logger_factory`.

It is a callable that returns the logger that gets wrapped and returned. In the simplest case, it's a function that returns a logger – or just a class. But you can also pass in an instance of a class with a `__call__` method for more complicated setups.

New in version 0.4.0: *structlog.get_logger()* can optionally take positional parameters.

These will be passed to the logger factories. For example, if you use run `structlog.get_logger('a name')` and configure structlog to use the standard library *LoggerFactory* which has support for positional parameters, the returned logger will have the name `'a name'`.

When writing custom logger factories, they should always accept positional parameters even if they don't use them. That makes sure that loggers are interchangeable.

For the common cases of standard library logging and Twisted logging, structlog comes with two factories built right in:

- *structlog.stdlib.LoggerFactory*
- *structlog.twisted.LoggerFactory*

So all it takes to use structlog with standard library logging is this:

```
>>> from structlog import get_logger, configure
>>> from structlog.stdlib import LoggerFactory
>>> configure(logger_factory=LoggerFactory())
>>> log = get_logger()
>>> log.critical('this is too easy!')
event='this is too easy!'
```

By using structlog's *structlog.stdlib.LoggerFactory*, it is also ensured that variables like function names and line numbers are expanded correctly in your log format.

The *Twisted example* shows how easy it is for Twisted.

Note: *LoggerFactory()*-style factories always need to get passed as *instances* like in the examples above. While neither allows for customization using parameters yet, they may do so in the future.

Calling *structlog.get_logger()* without configuration gives you a perfectly useful *structlog.PrintLogger* with the default values exaplained above. I don't believe silent loggers are a sensible default.

Where to Configure

The best place to perform your configuration varies with applications and frameworks. Ideally as late as possible but *before* non-framework (i.e. your) code is executed. If you use standard library's logging, it makes sense to configure them next to each other.

Django Django has to date unfortunately no concept of an application assembler or "app is done" hooks. Therefore the bottom of your `settings.py` will have to do.

Flask See Logging Application Errors.

Pyramid Application constructor.

Twisted The plugin definition is the best place. If your app is not a plugin, put it into your tac file (and then learn about plugins).

If you have no choice but *have* to configure on import time in module-global scope, or can't rule out for other reasons that that your `structlog.configure()` gets called more than once, structlog offers `structlog.configure_once()` that raises a warning if structlog has been configured before (no matter whether using `structlog.configure()` or `configure_once()`) but doesn't change anything.

2.1.5 Thread Local Context

Immutability

> You should call some functions with some arguments.
>
> —David Reid

The behavior of copying itself, adding new values, and returning the result is useful for applications that keep somehow their own context using classes or closures. Twisted is a *fine example* for that. Another possible approach is passing wrapped loggers around or log only within your view where you gather errors and events using return codes and exceptions. If you are willing to do that, you should stick to it because immutable state is a very good thing[3]. Sooner or later, global state and mutable data lead to unpleasant surprises.

However, in the case of conventional web development, we realize that passing loggers around seems rather cumbersome, intrusive, and generally against the mainstream culture. And since it's more important that people actually *use* structlog than to be pure and snobby, structlog contains a dirty but convenient trick: thread local context storage which you may already know from Flask:

Thread local storage makes your logger's context global but *only within the current thread*[4]. In the case of web frameworks this usually means that your context becomes global to the current request.

The following explanations may sound a bit confusing at first but the *Flask example* illustrates how simple and elegant this works in practice.

Wrapped Dicts

In order to make your context thread local, structlog ships with a function that can wrap any dict-like class to make it usable for thread local storage: `structlog.threadlocal.wrap_dict()`.

Within one thread, every instance of the returned class will have a *common* instance of the wrapped dict-like class:

[3] In the spirit of Python's 'consenting adults', structlog doesn't enforce the immutability with technical means. However, if you don't meddle with undocumented data, the objects can be safely considered immutable.

[4] Special care has been taken to detect and support greenlets properly.

```
>>> from structlog.threadlocal import wrap_dict
>>> WrappedDictClass = wrap_dict(dict)
>>> d1 = WrappedDictClass({'a': 1})
>>> d2 = WrappedDictClass({'b': 2})
>>> d3 = WrappedDictClass()
>>> d3['c'] = 3
>>> d1 is d3
False
>>> d1 == d2 == d3 == WrappedDictClass()
True
>>> d3
<WrappedDict-...({'a': 1, 'c': 3, 'b': 2})>
```

Then use an instance of the generated class as the context class:

```
configure(context_class=WrappedDictClass())
```

Note: **Remember**: the instance of the class *doesn't* matter. Only the class *type* matters because *all* instances of one class *share* the *same* data.

---

structlog.threadlocal.wrap_dict() returns always a completely *new* wrapped class:

```
>>> from structlog.threadlocal import wrap_dict
>>> WrappedDictClass = wrap_dict(dict)
>>> AnotherWrappedDictClass = wrap_dict(dict)
>>> WrappedDictClass() != AnotherWrappedDictClass()
True
>>> WrappedDictClass.__name__
WrappedDict-41e8382d-bee5-430e-ad7d-133c844695cc
>>> AnotherWrappedDictClass.__name__
WrappedDict-e0fc330e-e5eb-42ee-bcec-ffd7bd09ad09
```

In order to be able to bind values temporarily to a logger, *structlog.threadlocal* comes with a context manager: *tmp_bind()*:

```
>>> log.bind(x=42)
<BoundLogger(context=<WrappedDict-...({'x': 42})>, ...)>
>>> log.msg('event!')
x=42 event='event!'
>>> with tmp_bind(log, x=23, y='foo') as tmp_log:
...     tmp_log.msg('another event!')
y='foo' x=23 event='another event!'
>>> log.msg('one last event!')
x=42 event='one last event!'
```

The state before the `with` statement is saved and restored once it's left.

If you want to detach a logger from thread local data, there's *structlog.threadlocal.as_immutable()*.

Downsides & Caveats

The convenience of having a thread local context comes at a price though:

Warning:

- If you can't rule out that your application re-uses threads, you *must* remember to **initialize your thread local context** at the start of each request using `new()` (instead of `bind()`). Otherwise you may start a new request with the context still filled with data from the request before.
- **Don't** stop assigning the results of your `bind()`s and `new()`s!
  Do:

```
log = log.new(y=23)
log = log.bind(x=42)
```

Don't:

```
log.new(y=23)
log.bind(x=42)
```

Although the state is saved in a global data structure, you still need the global wrapped logger produce a real bound logger. Otherwise each log call will result in an instantiation of a temporary BoundLogger. See *Configuration* for more details.

The general sentiment against thread locals is that they're hard to test. In this case we feel like this is an acceptable trade-off. You can easily write deterministic tests using a call-capturing processor if you use the API properly (cf. warning above).

This big red box is also what separates immutable local from mutable global data.

2.1.6 Processors

The true power of `structlog` lies in its *combinable log processors*. A log processor is a regular callable, i.e. a function or an instance of a class with a `__call__()` method.

Chains

The *processor chain* is a list of processors. Each processors receives three positional arguments:

logger Your wrapped logger object. For example logging.Logger.

method_name The name of the wrapped method. If you called `log.warn('foo')`, it will be `"warn"`.

event_dict Current context together with the current event. If the context was `{'a': 42}` and the event is `"foo"`, the initial `event_dict` will be `{'a':42, 'event': 'foo'}`.

The return value of each processor is passed on to the next one as `event_dict` until finally the return value of the last processor gets passed into the wrapped logging method.

Examples

If you set up your logger like:

```
from structlog import PrintLogger, wrap_logger
wrapped_logger = PrintLogger()
logger = wrap_logger(wrapped_logger, processors=[f1, f2, f3, f4])
log = logger.new(x=42)
```

and call `log.msg('some_event', y=23)`, it results in the following call chain:

```
wrapped_logger.msg(
   f4(wrapped_logger, 'msg',
      f3(wrapped_logger, 'msg',
         f2(wrapped_logger, 'msg',
            f1(wrapped_logger, 'msg', {'event': 'some_event', 'x': 42, 'y': 23})
         )
      )
   )
)
```

In this case, `f4` has to make sure it returns something `wrapped_logger.msg` can handle (see *Adapting and Rendering*).

The simplest modification a processor can make is adding new values to the `event_dict`. Parsing human-readable timestamps is tedious, not so UNIX timestamps – let's add one to each log entry!

```
import calendar
import time


def timestamper(logger, log_method, event_dict):
    event_dict['timestamp'] = calendar.timegm(time.gmtime())
    return event_dict
```

Easy, isn't it? Please note, that structlog comes with such an processor built in: *`TimeStamper`*.

Filtering

If a processor raises *`structlog.DropEvent`*, the event is silently dropped.

Therefore, the following processor drops every entry:

```
from structlog import DropEvent


def dropper(logger, method_name, event_dict):
    raise DropEvent
```

But we can do better than that! How about dropping only log entries that are marked as coming from a certain peer (e.g. monitoring)?

```
from structlog import DropEvent


class ConditionalDropper(object):
    def __init__(self, peer_to_ignore):
        self._peer_to_ignore = peer_to_ignore

    def __call__(self, logger, method_name, event_dict):
        """
        >>> cd = ConditionalDropper('127.0.0.1')
        >>> cd(None, None, {'event': 'foo', 'peer': '10.0.0.1'})
        {'peer': '10.0.0.1', 'event': 'foo'}
        >>> cd(None, None, {'event': 'foo', 'peer': '127.0.0.1'})
        Traceback (most recent call last):
        ...
        DropEvent
        """
```

```
        if event_dict.get('peer') == self._peer_to_ignore:
            raise DropEvent
        else:
            return event_dict
```

Adapting and Rendering

An important role is played by the *last* processor because its duty is to adapt the `event_dict` into something the underlying logging method understands. With that, it's also the *only* processor that needs to know anything about the underlying system.

For that, it can either return a string that is passed as the first (and only) positional argument to the underlying logger or a tuple of `(args, kwargs)` that are passed as `log_method(*args, **kwargs)`. Therefore `return 'hello world'` is a shortcut for `return (('hello world',), {})` (the example in *Chains* assumes this shortcut has been taken).

This should give you enough power to use structlog with any logging system while writing agnostic processors that operate on dictionaries.

Examples

The probably most useful formatter for string based loggers is *`JSONRenderer`*. Advanced log aggregation and analysis tools like logstash offer features like telling them "this is JSON, deal with it" instead of fiddling with regular expressions.

More examples can be found in the *examples* chapter. For a list of shipped processors, check out the *API documentation*.

2.1.7 Examples

This chapter is intended to give you a taste of realistic usage of structlog.

Flask and Thread Local Data

In the simplest case, you bind a unique request ID to every incoming request so you can easily see which log entries belong to which request.

```
import uuid

import flask
import structlog

from .some_module import some_function


logger = structlog.get_logger()
app = flask.Flask(__name__)


@app.route('/login', methods=['POST', 'GET'])
def some_route():
    log = logger.new(
        request_id=str(uuid.uuid4()),
```

```
    )
    # do something
    # ...
    log.info('user logged in', user='test-user')
    # gives you:
    # event='user logged in' request_id='ffcdc44f-b952-4b5f-95e6-0f1f3a9ee5fd' user='test-user'
    # ...
    some_function()
    # ...

if __name__ == "__main__":
    structlog.configure(
        processors=[
            structlog.processors.KeyValueRenderer(
                key_order=['event', 'request_id'],
            ),
        ],
        context_class=structlog.threadlocal.wrap_dict(dict),
        logger_factory=structlog.stdlib.LoggerFactory(),
    )
    app.run()
```

some_module.py

```
from structlog import get_logger

logger = get_logger()


def some_function():
    # later then:
    logger.error('user did something', something='shot_in_foot')
    # gives you:
    # event='user did something 'request_id='ffcdc44f-b952-4b5f-95e6-0f1f3a9ee5fd' something='shot_i
```

While wrapped loggers are *immutable* by default, this example demonstrates how to circumvent that using a thread local dict implementation for context data for convenience (hence the requirement for using *new()* for re-initializing the logger).

Please note that *structlog.stdlib.LoggerFactory* is a totally magic-free class that just deduces the name of the caller's module and does a logging.getLogger(). with it. It's used by *structlog.get_logger()* to rid you of logging boilerplate in application code. If you prefer to name your standard library loggers explicitly, a positional argument to *get_logger()* gets passed to the factory and used as the name.

Twisted, and Logging Out Objects

If you prefer to log less but with more context in each entry, you can bind everything important to your logger and log it out with each log entry.

```
import sys
import uuid

import structlog
import twisted

from twisted.internet import protocol, reactor
```

```
logger = structlog.getLogger()


class Counter(object):
    i = 0

    def inc(self):
        self.i += 1

    def __repr__(self):
        return str(self.i)


class Echo(protocol.Protocol):
    def connectionMade(self):
        self._counter = Counter()
        self._log = logger.new(
            connection_id=str(uuid.uuid4()),
            peer=self.transport.getPeer().host,
            count=self._counter,
        )

    def dataReceived(self, data):
        self._counter.inc()
        log = self._log.bind(data=data)
        self.transport.write(data)
        log.msg('echoed data!')


if __name__ == "__main__":
    structlog.configure(
        processors=[structlog.twisted.EventAdapter()],
        logger_factory=structlog.twisted.LoggerFactory(),
    )
    twisted.python.log.startLogging(sys.stderr)
    reactor.listenTCP(1234, protocol.Factory.forProtocol(Echo))
    reactor.run()
```

gives you something like:

```
... peer='127.0.0.1' connection_id='1c6c0cb5-...' count=1 data='123\n' event='echoed data!'
... peer='127.0.0.1' connection_id='1c6c0cb5-...' count=2 data='456\n' event='echoed data!'
... peer='127.0.0.1' connection_id='1c6c0cb5-...' count=3 data='foo\n' event='echoed data!'
... peer='10.10.0.1' connection_id='85234511-...' count=1 data='cba\n' event='echoed data!'
... peer='127.0.0.1' connection_id='1c6c0cb5-...' count=4 data='bar\n' event='echoed data!'
```

Since Twisted's logging system is a bit peculiar, structlog ships with an *adapter* so it keeps behaving like you'd expect it to behave.

I'd also like to point out the Counter class that doesn't do anything spectacular but gets bound *once* per connection to the logger and since its repr is the number itself, it's logged out correctly for each event. This shows off the strength of keeping a dict of objects for context instead of passing around serialized strings.

Processors

Processors are a both simple and powerful feature of structlog.

So you want timestamps as part of the structure of the log entry, censor passwords, filter out log entries below your

log level before they even get rendered, and get your output as JSON for convenient parsing? Here you go:

```
>>> import datetime, logging, sys
>>> from structlog import wrap_logger
>>> from structlog.processors import JSONRenderer
>>> from structlog.stdlib import filter_by_level
>>> logging.basicConfig(stream=sys.stdout, format='%(message)s')
>>> def add_timestamp(_, __, event_dict):
...     event_dict['timestamp'] = datetime.datetime.utcnow()
...     return event_dict
>>> def censor_password(_, __, event_dict):
...     pw = event_dict.get('password')
...     if pw:
...         event_dict['password'] = '*CENSORED*'
...     return event_dict
>>> log = wrap_logger(
...     logging.getLogger(__name__),
...     processors=[
...         filter_by_level,
...         add_timestamp,
...         censor_password,
...         JSONRenderer(indent=1, sort_keys=True)
...     ]
... )
>>> log.info('something.filtered')
>>> log.warning('something.not_filtered', password='secret')
{
"event": "something.not_filtered",
"password": "*CENSORED*",
"timestamp": "datetime.datetime(..., ..., ..., ..., ...)"
}
```

structlog comes with many handy processors build right in – for a list of shipped processors, check out the *API documentation*.

2.2 Integration with Existing Systems

structlog can be used immediately with any existing logger. However it comes with special wrappers for the Python standard library and Twisted that are optimized for their respective underlying loggers and contain less magic.

2.2.1 Python Standard Library

Concrete Bound Logger

To make structlog's behavior less magicy, it ships with a standard library-specific wrapper class that has an explicit API instead of improvising: *`structlog.stdlib.BoundLogger`*. It behaves exactly like the generic *`structlog.BoundLogger`* except:

- it's slightly faster due to less overhead,
- has an explicit API that mirrors the log methods of standard library's Logger,
- hence causing less cryptic error messages if you get method names wrong.

Processors

structlog comes with one standard library-specific processor:

***filter_by_level()*:** Checks the log entries's log level against the configuration of standard library's logging. Log entries below the threshold get silently dropped. Put it at the beginning of your processing chain to avoid expensive operations happen in the first place.

Suggested Configuration

```
import structlog

structlog.configure(
   processors=[
        structlog.stdlib.filter_by_level,
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.JSONRenderer()
   ],
   context_class=dict,
   logger_factory=structlog.stdlib.LoggerFactory(),
   wrapper_class=structlog.stdlib.BoundLogger,
   cache_logger_on_first_use=True,
)
```

See also Logging Best Practices.

2.2.2 Twisted

Warning: Currently, the Twisted-specific code is *not* tested against Python 3.3. This is caused by this Twisted bug and will remedied once that bug is fixed.

Concrete Bound Logger

To make structlog's behavior less magicy, it ships with a Twisted-specific wrapper class that has an explicit API instead of improvising: *structlog.twisted.BoundLogger*. It behaves exactly like the generic *structlog.BoundLogger* except:

- it's slightly faster due to less overhead,
- has an explicit API (*msg()* and *err()*),
- hence causing less cryptic error messages if you get method names wrong.

In order to avoid that structlog disturbs your CamelCase harmony, it comes with an alias for *structlog.get_logger()* called *structlog.getLogger()*.

Processors

structlog comes with two Twisted-specific processors:

EventAdapter This is useful if you have an existing Twisted application and just want to wrap your loggers for now. It takes care of transforming your event dictionary into something twisted.python.log.err can digest.

For example:

```
def onError(fail):
   failure = fail.trap(MoonExploded)
   log.err(failure, _why='event-that-happend')
```

will still work as expected.

Needs to be put at the end of the processing chain. It formats the event using a renderer that needs to be passed into the constructor:

```
configure(processors=[EventAdapter(KeyValueRenderer()])
```

The drawback of this approach is that Twisted will format your exceptions as multi-line log entries which is painful to parse. Therefore structlog comes with:

JSONRenderer Goes a step further and circumvents Twisted logger's Exception/Failure handling and renders it itself as JSON strings. That gives you regular and simple-to-parse single-line JSON log entries no matter what happens.

Bending Foreign Logging To Your Will

structlog comes with a wrapper for Twisted's log observers to ensure the rest of your logs are in JSON too: *JSONLogObserverWrapper()*.

What it does is determining whether a log entry has been formatted by *JSONRenderer* and if not, converts the log entry to JSON with *event* being the log message and putting Twisted's *system* into a second key.

So for example:

```
2013-09-15 22:02:18+0200 [-] Log opened.
```

becomes:

```
2013-09-15 22:02:18+0200 [-] {"event": "Log opened.", "system": "-"}
```

There is obviously some redundancy here. Also, I'm presuming that if you write out JSON logs, you're going to let something else parse them which makes the human-readable date entries more trouble than they're worth.

To get a clean log without timestamps and additional system fields (`[-]`), structlog comes with *PlainFileLogObserver* that writes only the plain message to a file and *plainJSONStdOutLogger()* that composes it with the aforementioned *JSONLogObserverWrapper()* and gives you a pure JSON log without any timestamps or other noise straight to standard out:

```
$ twistd -n --logger structlog.twisted.plainJSONStdOutLogger web
{"event": "Log opened.", "system": "-"}
{"event": "twistd 13.1.0 (python 2.7.3) starting up.", "system": "-"}
{"event": "reactor class: twisted...EPollReactor.", "system": "-"}
{"event": "Site starting on 8080", "system": "-"}
{"event": "Starting factory <twisted.web.server.Site ...>", ...}
...
```

Suggested Configuration

```
import structlog

structlog.configure(
   processors=[
```

```
        structlog.processors.StackInfoRenderer(),
        structlog.twisted.JSONRenderer()
    ],
    context_class=dict,
    logger_factory=structlog.twisted.LoggerFactory(),
    wrapper_class=structlog.twisted.BoundLogger,
    cache_logger_on_first_use=True,
)
```

See also Logging Best Practices.

2.2.3 Logging Best Practices

The best practice for you depends very much on your context. To give you some pointers nevertheless, here are a few scenarios that may be applicable to you.

Pull requests for further interesting approaches as well as refinements and more complete examples are very welcome.

Common Ideas

Logging is not a new concept and in no way special to Python. Logfiles have existed for decades and there's little reason to reinvent the wheel in our little world.

There are several concepts that are very well-solved in general and especially in heterogeneous environments, using special tooling for Python applications does more harm than good and makes the operations staff build dart board with your pictures.

Therefore let's rely on proven tools as much as possible and do only the absolutely necessary inside of Python[5]. A very nice approach is to simply log to standard out and let other tools take care of the rest.

runit

One runner that makes this very easy is the venerable runit project which made it a part of its design: server processes don't detach but log to standard out instead. There it gets processed by other software – potentially by one of its own tools: svlogd. We use it extensively and it has proven itself extremely robust and capable; check out this tutorial if you'd like to try it.

If you're not quite convinced and want an overview on running daemons, have a look at cue's daemon showdown that discusses the most common ones.

Local Logging

There are basically two common ways to log to local logfiles: writing yourself into files and syslog.

Syslog

The simplest approach to logging is to forward your entries to the syslogd. Twisted, uwsgi, and runit support it directly. It will happily add a timestamp and write wherever you tell it in its configuration. You can also log from multiple processes into a single file and use your system's logrotate for log rotation.

[5] This is obviously a privileged UNIX-centric view but even Windows has tools and means for log management although we won't be able to discuss them here.

The only downside is that syslog has some quirks that show itself under high load like rate limits (they can be switched off) and lost log entries.

runit's svlogd

If you'll choose runit for running your daemons, svlogd is a nicer approach. It receives the log entries via a UNIX pipe and acts on them which includes adding of parse-friendly timestamps in tai64n as well as filtering and log rotation.

Centralized Logging

Nowadays you usually don't want your logfiles in compressed archives distributed over dozens – if not thousands – servers. You want them at a single location; parsed and easy to query.

Syslog (Again!)

The widely deployed syslog implementation rsyslog supports remote logging out-of-the-box. Have a look at this post by Revolution Systems on the how.

Since syslog is such a widespread solution, there are also ways to use it with basically any centralized product.

Logstash with logstash-forwarder

Logstash is a great way to parse, save, and search your logs.

The general modus operandi is that you have log shippers that parse your log files and forward the log entries to your Logstash server and store is in elasticsearch. If your log entries consist – as suggested – of a tai64n timestamp and a JSON dictionary, this is pretty easy and efficient.

If you can't decide on a log shipper, logstash-forwarder (formerly known as Lumberjack) works really well.

Graylog2

Graylog goes one step further. It not only supports everything those above do (and then some); you can also log directly JSON entries towards it – optionally even through an AMQP server (like RabbitMQ) for better reliability. Additionally, Graylog's Extended Log Format (GELF) allows for structured data which makes it an obvious choice to use together with structlog.

2.3 Advanced Topics

2.3.1 Custom Wrappers

structlog comes with a generic bound logger called *structlog.BoundLogger* that can be used to wrap any logger class you fancy. It does so by intercepting unknown method names and proxying them to the wrapped logger.

This works fine, except that it has a performance penalty and the API of *BoundLogger* isn't clear from reading the documentation because large parts depend on the wrapped logger. An additional reason is that you may want to have semantically meaningful log method names that add meta data to log entries as it is fit (see example below).

To solve that, structlog offers you to use an own wrapper class which you can configure using *structlog.configure()*. And to make it easier for you, it comes with the class

structlog.BoundLoggerBase which takes care of all data binding duties so you just add your log methods if you choose to sub-class it.

Example

It's much easier to demonstrate with an example:

```
>>> from structlog import BoundLoggerBase, PrintLogger, wrap_logger
>>> class SemanticLogger(BoundLoggerBase):
...     def msg(self, event, **kw):
...         if not 'status' in kw:
...             return self._proxy_to_logger('msg', event, status='ok', **kw)
...         else:
...             return self._proxy_to_logger('msg', event, **kw)
...
...     def user_error(self, event, **kw):
...         self.msg(event, status='user_error', **kw)
>>> log = wrap_logger(PrintLogger(), wrapper_class=SemanticLogger)
>>> log = log.bind(user='fprefect')
>>> log.user_error('user.forgot_towel')
user='fprefect' status='user_error' event='user.forgot_towel'
```

You can observe the following:

- The wrapped logger can be found in the instance variable `structlog.BoundLoggerBase._logger`.
- The helper method *structlog.BoundLoggerBase._proxy_to_logger()* that is a DRY convenience function that runs the processor chain, handles possible *DropEvent*s and calls a named function on `_logger`.
- You can run the chain by hand though using *structlog.BoundLoggerBase._process_event()* .

These two methods and one attribute is all you need to write own wrapper classes.

2.3.2 Performance

structlog's default configuration tries to be as unsurprising and not confusing to new developers as possible. Some of the choices made come with an avoidable performance price tag – although its impact is debatable.

Here are a few hints how to get most out of structlog in production:

1. Use plain *dict*s as context classes. Python is full of them and they are highly optimized:

   ```
   configure(context_class=dict)
   ```

   If you don't use automated parsing (you should!) and need predicable order of your keys for some reason, use the *key_order* argument of *KeyValueRenderer*.

2. Use a specific wrapper class instead of the generic one. structlog comes with ones for the Python Standard Library and for Twisted:

   ```
   configure(wrapper_class=structlog.stdlib.BoundLogger)
   ```

   Writing own wrapper classes is straightforward too.

3. Avoid (frequently) calling log methods on loggers you get back from *structlog.wrap_logger()* and *structlog.get_logger()*. Since those functions are usually called in module scope and thus before you are able to configure them, they return a proxy that assembles the correct logger on demand.

   Create a local logger if you expect to log frequently without binding:

```
logger = structlog.get_logger()
def f():
   log = logger.bind()
   for i in range(1000000000):
      log.info('iterated', i=i)
```

4. Set the *cache_logger_on_first_use* option to *True* so the aforementioned on-demand loggers will be assembled only once and cached for future uses:

```
configure(cache_logger_on_first_use=True)
```

   This has the only drawback is that later calls on *configure()* don't have any effect on already cached loggers – that shouldn't matter outside of testing though.

CHAPTER 3

Project Information

3.1 How To Contribute

Every open source project lives from the generous help by contributors that sacrifice their time and structlog is no different.

To make participation as pleasant as possible, this project adheres to the Code of Conduct by the Python Software Foundation.

Here are a few hints and rules to get you started:

- Add yourself to the AUTHORS.rst file in an alphabetical fashion. Every contribution is valuable and shall be credited.
- If your change is noteworthy, add an entry to the changelog.
- No contribution is too small; please submit as many fixes for typos and grammar bloopers as you can!
- Don't *ever* break backward compatibility. `structlog` is an infrastructure library people rely on; therefore highest care must be put into avoiding breakage on updates. If it ever *has* to happen for higher reasons, structlog will follow the proven procedures of the Twisted project.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation won't be merged – if a feature is not tested or documented, it doesn't exist.
- Obey PEP 8 and PEP 257. Twisted-specific modules use CamelCase.
- Write good commit messages.
- Ideally, squash your commits, i.e. make your pull requests just one commit.

Note: If you have something great but aren't sure whether it adheres – or even can adhere – to the rules above: **please submit a pull request anyway**!

In the best case, we can mold it into something, in the worst case the pull request gets politely closed. There's absolutely nothing to fear.

Thank you for considering to contribute to structlog! If you have any question or concerns, feel free to reach out to me – there is also a `#structlog` channel on freenode.

3.2 License and Hall of Fame

structlog is licensed under the permissive Apache License, Version 2. The full license text can be also found in the source code repository.

3.2.1 Authors

structlog is written and maintained by Hynek Schlawack. It's inspired by previous work done by Jean-Paul Calderone and David Reid.

The development is kindly supported by Variomedia AG.

The following folks helped forming structlog into what it is now:

- Alex Gaynor
- Christopher Armstrong
- Daniel Lindsley
- David Reid
- Donald Stufft
- George-Cristian Bîrzan
- Glyph
- Holger Krekel
- Itamar Turner-Trauring
- Jack Pearkes
- Jean-Paul Calderone
- Lynn Root
- Noah Kantrowitz
- Tarek Ziadé
- Thomas Heinrichsdobler
- Tom Lazar

Some of them disapprove of the addition of thread local context data. :)

Third Party Code

The compatibility code that makes this software run on both Python 2 and 3 is heavily inspired and partly copy and pasted from the MIT-licensed six by Benjamin Peterson. The only reason why it's not used as a dependency is to avoid any runtime dependency in the first place.

3.3 Changelog

- : `from structlog import *` works now (but you still shouldn't use it).
- : Fixed a memory leak in greenlet code that emulates thread locals. It shouldn't matter in practice unless you use multiple wrapped dicts within one program that is rather unlikely.

- : Various doc fixes.
- : Don't cache proxied methods in `structlog.threadlocal._ThreadLocalDictWrapper`. This doesn't affect regular users.
- #5: Add meta data (e.g. function names, line numbers) extraction for wrapped stdlib loggers.
- : Allow the standard library name guesser to ignore certain frame names. This is useful together with frameworks.
- : Add *structlog.processors.ExceptionPrettyPrinter* for development and testing when multiline log entries aren't just acceptable but even helpful.
- #12: Allow optional positional arguments for *structlog.get_logger()* that are passed to logger factories. The standard library factory uses this for explicit logger naming.
- #6: Add *structlog.processors.StackInfoRenderer* for adding stack information to log entries without involving exceptions. Also added it to default processor chain.
- : Fix stdlib's name guessing.
- : Add forgotten *structlog.processors.TimeStamper* to API documentation.
- : Extract a common base class for loggers that does nothing except keeping the context state. This makes writing custom loggers much easier and more straight-forward. See *structlog.BoundLoggerBase*.
- : Allow logger proxies that are returned by *structlog.get_logger()* and *structlog.wrap_logger()* to cache the BoundLogger they assemble according to configuration on first use. See Performance and the *cache_logger_on_first_use* of *structlog.configure()* and *structlog.wrap_logger()*.
- : Add Twisted-specific BoundLogger that has an explicit API instead of intercepting unknown method calls. See *structlog.twisted.BoundLogger*.
- : *structlog.ReturnLogger* now allows arbitrary positional and keyword arguments.
- : Add Python Standard Library-specific BoundLogger that has an explicit API instead of intercepting unknown method calls. See *structlog.stdlib.BoundLogger*.
- : Greatly enhanced and polished the documentation and added a new theme based on Write The Docs, requests, and Flask. See License and Hall of Fame.
- : Allow for custom serialization in *structlog.twisted.JSONRenderer* without abusing `__repr__`.
- : Enhance Twisted support by offering JSONification of non-structlog log entries.
- : *structlog.PrintLogger* now uses proper I/O routines and is thus viable not only for examples but also for production.
- : Add *key_order* option to *structlog.processors.KeyValueRenderer* for more predictable log entries with any *dict* class.
- : Promote to stable, thus henceforth a strict backward compatibility policy is put into effect. See *How To Contribute*.
- : Initial work.

CHAPTER 4

API Reference

4.1 structlog Package

4.1.1 `structlog` Package

`structlog.`**`get_logger`**(**args*, ***initial_values*)

Convenience function that returns a logger according to configuration.

```
>>> from structlog import get_logger
>>> log = get_logger(y=23)
>>> log.msg('hello', x=42)
y=23 x=42 event='hello'
```

Parameters

- **`args`** – *Optional* positional arguments that are passed unmodified to the logger factory. Therefore it depends on the factory what they mean.
- **`initial_values`** – Values that are used to pre-populate your contexts.

Return type A proxy that creates a correctly configured bound logger when necessary.

See *Configuration* for details.

If you prefer CamelCase, there's an alias for your reading pleasure: *`structlog.getLogger()`*.

New in version 0.4.0: *args*

`structlog.`**`getLogger`**(**args*, ***initial_values*)

Convenience function that returns a logger according to configuration.

```
>>> from structlog import get_logger
>>> log = get_logger(y=23)
>>> log.msg('hello', x=42)
y=23 x=42 event='hello'
```

Parameters

- **`args`** – *Optional* positional arguments that are passed unmodified to the logger factory. Therefore it depends on the factory what they mean.
- **`initial_values`** – Values that are used to pre-populate your contexts.

Return type A proxy that creates a correctly configured bound logger when necessary.

See *Configuration* for details.

If you prefer CamelCase, there's an alias for your reading pleasure: `structlog.getLogger()`.

New in version 0.4.0: *args*

`structlog.`**`wrap_logger`**(*logger*, *processors=None*, *wrapper_class=None*, *context_class=None*, *cache_logger_on_first_use=None*, *logger_factory_args=None*, ***initial_values*)

Create a new bound logger for an arbitrary *logger*.

Default values for *processors*, *wrapper_class*, and *context_class* can be set using `configure()`.

If you set an attribute here, `configure()` calls have *no* effect for the *respective* attribute.

In other words: selective overwriting of the defaults while keeping some *is* possible.

Parameters

- **`initial_values`** – Values that are used to pre-populate your contexts.
- **`logger_factory_args`** (*tuple*) – Values that are passed unmodified as `*logger_factory_args` to the logger factory if not *None*.

Return type A proxy that creates a correctly configured bound logger when necessary.

See `configure()` for the meaning of the rest of the arguments.

New in version 0.4.0: *logger_factory_args*

`structlog.`**`configure`**(*processors=None*, *wrapper_class=None*, *context_class=None*, *logger_factory=None*, *cache_logger_on_first_use=None*)

Configures the **global** defaults.

They are used if `wrap_logger()` has been called without arguments.

Also sets the global class attribute `is_configured` to *True* on first call. Can be called several times, keeping an argument at *None* leaves is unchanged from the current setting.

Use `reset_defaults()` to undo your changes.

Parameters

- **`processors`** (*list*) – List of processors.
- **`wrapper_class`** (*type*) – Class to use for wrapping loggers instead of `structlog.BoundLogger`. See Python Standard Library, Twisted, and Custom Wrappers.
- **`context_class`** (*type*) – Class to be used for internal context keeping.
- **`logger_factory`** (*callable*) – Factory to be called to create a new logger that shall be wrapped.
- **`cache_logger_on_first_use`** (*bool*) – *wrap_logger* doesn't return an actual wrapped logger but a proxy that assembles one when it's first used. If this option is set to *True*, this assembled logger is cached. See Performance.

New in version 0.3.0: *cache_logger_on_first_use*

`structlog.`**`configure_once`**(**args*, ***kw*)

Configures iff structlog isn't configured yet.

It does *not* matter whether is was configured using `configure()` or `configure_once()` before.

Raises a RuntimeWarning if repeated configuration is attempted.

structlog.**reset_defaults**()
Resets global default values to builtins.

That means [*StackInfoRenderer*, *format_exc_info()*, *KeyValueRenderer*] for *processors*, *BoundLogger* for *wrapper_class*, OrderedDict for *context_class*, *PrintLoggerFactory* for *logger_factory*, and *False* for *cache_logger_on_first_use*.

Also sets the global class attribute is_configured to *False*.

class structlog.**BoundLogger**(*logger*, *processors*, *context*)
A generic BoundLogger that can wrap anything.

Every unknown method will be passed to the wrapped logger. If that's too much magic for you, try *structlog.twisted.BoundLogger* or *:class:'structlog.twisted.BoundLogger* which also take advantage of knowing the wrapped class which generally results in better performance.

Not intended to be instantiated by yourself. See *wrap_logger()* and *get_logger()*.

bind(***new_values*)
Return a new logger with *new_values* added to the existing ones.

Return type *self.__class__*

new(***new_values*)
Clear context and binds *initial_values* using *bind()*.

Only necessary with dict implementations that keep global state like those wrapped by *structlog.threadlocal.wrap_dict()* when threads are re-used.

Return type *self.__class__*

unbind(**keys*)
Return a new logger with *keys* removed from the context.

Raises KeyError If the key is not part of the context.

Return type *self.__class__*

class structlog.**PrintLogger**(*file=None*)
Prints events into a file.

Parameters **file** (*file*) – File to print to. (default: stdout)

```
>>> from structlog import PrintLogger
>>> PrintLogger().msg('hello')
hello
```

Useful if you just capture your stdout with tools like runit or if you forward your stderr to syslog.

Also very useful for testing and examples since logging is sometimes finicky in doctests.

critical(*message*)
Print *message*.

debug(*message*)
Print *message*.

err(*message*)
Print *message*.

error(*message*)
Print *message*.

info(*message*)
Print *message*.

log(*message*)
Print *message*.

msg(*message*)
Print *message*.

warning(*message*)
Print *message*.

class structlog.**PrintLoggerFactory**(*file=None*)
Produces *PrintLogger*s.

To be used with *structlog.configure()*'s *logger_factory*.

Parameters **file** (*file*) – File to print to. (default: stdout)

Positional arguments are silently ignored.

New in version 0.4.0.

class structlog.**ReturnLogger**
Returns the string that it's called with.

```
>>> from structlog import ReturnLogger
>>> ReturnLogger().msg('hello')
'hello'
>>> ReturnLogger().msg('hello', when='again')
(('hello',), {'when': 'again'})
```

Useful for unit tests.

Changed in version 0.3.0: Allow for arbitrary arguments and keyword arguments to be passed in.

critical(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

debug(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

err(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

error(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

info(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

log(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

msg(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

warning(**args*, ***kw*)
Return tuple of `args, kw` or just `args[0]` if only one arg passed

class structlog.**ReturnLoggerFactory**
Produces and caches *ReturnLogger*s.

To be used with *structlog.configure()*'s *logger_factory*.

Positional arguments are silently ignored.

New in version 0.4.0.

exception `structlog.`**`DropEvent`**
If raised by an processor, the event gets silently dropped.

Derives from BaseException because it's technically not an error.

class `structlog.`**`BoundLoggerBase`**(*logger*, *processors*, *context*)
Immutable context carrier.

Doesn't do any actual logging; examples for useful subclasses are:

- the generic *BoundLogger* that can wrap anything,
- *structlog.twisted.BoundLogger*,
- and *structlog.stdlib.BoundLogger*.

See also Custom Wrappers.

`_process_event`(*method_name*, *event*, *event_kw*)
Combines creates an *event_dict* and runs the chain.

Call it to combine your *event* and *context* into an event_dict and process using the processor chain.

Parameters

- **`method_name`** (*str*) – The name of the logger method. Is passed into the processors.
- **`event`** – The event – usually the first positional argument to a logger.
- **`event_kw`** – Additional event keywords. For example if someone calls `log.msg('foo', bar=42)`, *event* would to be `'foo'` and *event_kw* `{'bar': 42}`.

Raises *structlog.DropEvent* if log entry should be dropped.

Return type *tuple* of *(*args, **kw)*

Note: Despite underscore available to custom wrapper classes.

See also Custom Wrappers.

`_proxy_to_logger`(*method_name*, *event=None*, ***event_kw*)
Run processor chain on event & call *method_name* on wrapped logger.

DRY convenience method that runs *_process_event()*, takes care of handling *structlog.DropEvent*, and finally calls *method_name* on `_logger` with the result.

Parameters

- **`method_name`** (*str*) – The name of the method that's going to get called. Technically it should be identical to the method the user called because it also get passed into processors.
- **`event`** – The event – usually the first positional argument to a logger.
- **`event_kw`** – Additional event keywords. For example if someone calls `log.msg('foo', bar=42)`, *event* would to be `'foo'` and *event_kw* `{'bar': 42}`.

Note: Despite underscore available to custom wrapper classes.

See also Custom Wrappers.

`bind`(***new_values*)
Return a new logger with *new_values* added to the existing ones.

Return type *self.__class__*

new(***new_values*)
Clear context and binds *initial_values* using `bind()`.

Only necessary with dict implementations that keep global state like those wrapped by `structlog.threadlocal.wrap_dict()` when threads are re-used.

Return type *self.__class__*

unbind(**keys*)
Return a new logger with *keys* removed from the context.

Raises KeyError If the key is not part of the context.

Return type *self.__class__*

4.1.2 threadlocal Module

Primitives to keep context global but thread (and greenlet) local.

structlog.threadlocal.**wrap_dict**(*dict_class*)
Wrap a dict-like class and return the resulting class.

The wrapped class and used to keep global in the current thread.

Parameters **dict_class** (*type*) – Class used for keeping context.

Return type *type*

structlog.threadlocal.**tmp_bind**(*logger*, ***tmp_values*)
Bind *tmp_values* to *logger* & memorize current state. Rewind afterwards.

```
>>> from structlog import wrap_logger, PrintLogger
>>> from structlog.threadlocal import tmp_bind, wrap_dict
>>> logger = wrap_logger(PrintLogger(),  context_class=wrap_dict(dict))
>>> with tmp_bind(logger, x=5) as tmp_logger:
...     logger = logger.bind(y=3)
...     tmp_logger.msg('event')
y=3 x=5 event='event'
>>> logger.msg('event')
event='event'
```

structlog.threadlocal.**as_immutable**(*logger*)
Extract the context from a thread local logger into an immutable logger.

Parameters **logger** (BoundLogger) – A logger with *possibly* thread local state.

Return type `BoundLogger` with an immutable context.

4.1.3 processors Module

Processors useful regardless of the logging framework.

class structlog.processors.**JSONRenderer**(***dumps_kw*)
Render the *event_dict* using *json.dumps(event_dict, **json_kw)*.

Parameters **json_kw** – Are passed unmodified to *json.dumps()*.

```
>>> from structlog.processors import JSONRenderer
>>> JSONRenderer(sort_keys=True)(None, None, {'a': 42, 'b': [1, 2, 3]})
'{"a": 42, "b": [1, 2, 3]}'
```

Bound objects are attempted to be serialize using a `__structlog__` method. If none is defined, `repr()` is used:

```
>>> class C1(object):
...     def __structlog__(self):
...         return ['C1!']
...     def __repr__(self):
...         return '__structlog__ took precedence'
>>> class C2(object):
...     def __repr__(self):
...         return 'No __structlog__, so this is used.'
>>> from structlog.processors import JSONRenderer
>>> JSONRenderer(sort_keys=True)(None, None, {'c1': C1(), 'c2': C2()})
'{"c1": ["C1!"], "c2": "No __structlog__, so this is used."}'
```

Please note that additionally to strings, you can also return any type the standard library JSON module knows about – like in this example a list.

Changed in version 0.2.0: Added support for `__structlog__` serialization method.

class `structlog.processors.`**`KeyValueRenderer`**(*sort_keys=False*, *key_order=None*)
Render *event_dict* as a list of `Key=repr(Value)` pairs.

Parameters

- **`sort_keys`** (*bool*) – Whether to sort keys when formatting.
- **`key_order`** (*list*) – List of keys that should be rendered in this exact order. Missing keys will be rendered as *None*, extra keys depending on *sort_keys* and the dict class.

```
>>> from structlog.processors import KeyValueRenderer
>>> KeyValueRenderer(sort_keys=True)(None, None, {'a': 42, 'b': [1, 2, 3]})
'a=42 b=[1, 2, 3]'
>>> KeyValueRenderer(key_order=['b', 'a'])(None, None,
...                                        {'a': 42, 'b': [1, 2, 3]})
'b=[1, 2, 3] a=42'
```

New in version 0.2.0: *key_order*

class `structlog.processors.`**`UnicodeEncoder`**(*encoding='utf-8'*, *errors='backslashreplace'*)
Encode unicode values in *event_dict*.

Parameters

- **`encoding`** (*str*) – Encoding to encode to (default: `'utf-8'`.
- **`errors`** (*str*) – How to cope with encoding errors (default `'backslashreplace'`).

Useful for *KeyValueRenderer* if you don't want to see u-prefixes:

```
>>> from structlog.processors import KeyValueRenderer, UnicodeEncoder
>>> KeyValueRenderer()(None, None, {'foo': u'bar'})
"foo=u'bar'"
>>> KeyValueRenderer()(None, None,
...                    UnicodeEncoder()(None, None, {'foo': u'bar'}))
"foo='bar'"
```

or *JSONRenderer* and *structlog.twisted.JSONRenderer* to make sure user-supplied strings don't break the renderer.

Just put it in the processor chain before the renderer.

structlog.processors.**format_exc_info**(*logger*, *name*, *event_dict*)
Replace an *exc_info* field by an *exception* string field:

If *event_dict* contains the key `exc_info`, there are two possible behaviors:

- If the value is a tuple, render it into the key `exception`.
- If the value true but no tuple, obtain exc_info ourselves and render that.

If there is no `exc_info` key, the *event_dict* is not touched. This behavior is analogue to the one of the stdlib's logging.

```
>>> from structlog.processors import format_exc_info
>>> try:
...     raise ValueError
... except ValueError:
...     format_exc_info(None, None, {'exc_info': True})
{'exception': 'Traceback (most recent call last):...
```

class structlog.processors.**StackInfoRenderer**
Add stack information with key *stack* if *stack_info* is true.

Useful when you want to attach a stack dump to a log entry without involving an exception.

It works analogously to the *stack_info* argument of the Python 3 standard library logging but works on both 2 and 3.

New in version 0.4.0.

class structlog.processors.**ExceptionPrettyPrinter**(*file=None*)
Pretty print exceptions and remove them from the *event_dict*.

Parameters **file** (*file*) – Target file for output (default: *sys.stdout*).

This processor is mostly for development and testing so you can read exceptions properly formatted.

It behaves like *format_exc_info()* except it removes the exception data from the event dictionary after printing it.

It's tolerant to having *format_exc_info* in front of itself in the processor chain but doesn't require it. In other words, it handles both *exception* as well as *exc_info* keys.

New in version 0.4.0.

class structlog.processors.**TimeStamper**(*fmt=None*, *utc=True*)
Add a timestamp to *event_dict*.

Note: You probably want to let OS tools take care of timestamping. See also Logging Best Practices.

Parameters

- **format** (*str*) – strftime format string, or `"iso"` for ISO 8601, or *None* for a UNIX timestamp.
- **utc** (*bool*) – Whether timestamp should be in UTC or local time.

```
>>> from structlog.processors import TimeStamper
>>> TimeStamper()(None, None, {})
{'timestamp': 1378994017}
>>> TimeStamper(fmt='iso')(None, None, {})
{'timestamp': '2013-09-12T13:54:26.996778Z'}
```

```
>>> TimeStamper(fmt='%Y')(None, None, {})
{'timestamp': '2013'}
```

4.1.4 `stdlib` Module

Processors and helpers specific to the logging module from the Python standard library.

class `structlog.stdlib.`**`BoundLogger`**(*logger*, *processors*, *context*)
Python Standard Library version of *`structlog.BoundLogger`*. Works exactly like the generic one except that it takes advantage of knowing the logging methods in advance.

Use it like:

```
configure(
    wrapper_class=structlog.stdlib.BoundLogger,
)
```

class `structlog.stdlib.`**`LoggerFactory`**(*ignore_frame_names=None*)
Build a standard library logger when an *instance* is called.

Sets a custom logger using *logging.setLogggerClass* so variables in log format are expanded properly.

```
>>> from structlog import configure
>>> from structlog.stdlib import LoggerFactory
>>> configure(logger_factory=LoggerFactory())
```

Parameters **`ignore_frame_names`** (*list* of *str*) – When guessing the name of a logger, skip frames whose names *start* with one of these. For example, in pyramid applications you'll want to set it to `['venusian', 'pyramid.config']`.

`__call__`(**args*)
Deduce the caller's module name and create a stdlib logger.

If an optional argument is passed, it will be used as the logger name instead of guesswork. This optional argument would be passed from the *`structlog.get_logger()`* call. For example `struclog.get_logger('foo')` would cause this method to be called with `'foo'` as its first positional argument.

Return type *logging.Logger*

Changed in version 0.4.0: Added support for optional positional arguments. Using the first one for naming the constructed logger.

`structlog.stdlib.`**`filter_by_level`**(*logger*, *name*, *event_dict*)
Check whether logging is configured to accept messages from this log level.

Should be the first processor if stdlib's filtering by level is used so possibly expensive processors like exception formatters are avoided in the first place.

```
>>> import logging
>>> from structlog.stdlib import filter_by_level
>>> logging.basicConfig(level=logging.WARN)
>>> logger = logging.getLogger()
>>> filter_by_level(logger, 'warn', {})
{}
>>> filter_by_level(logger, 'debug', {})
Traceback (most recent call last):
...
DropEvent
```

4.1.5 `twisted` Module

Processors and tools specific to the Twisted networking engine.

See also structlog's Twisted support.

class `structlog.twisted.`**`BoundLogger`**(*logger*, *processors*, *context*)
Twisted-specific version of *`structlog.BoundLogger`*.

Works exactly like the generic one except that it takes advantage of knowing the logging methods in advance.

Use it like:

```
configure(
    wrapper_class=structlog.twisted.BoundLogger,
)
```

`bind`(***new_values*)
Return a new logger with *new_values* added to the existing ones.

Return type *self.__class__*

`err`(*event=None*, ***kw*)
Process event and call `log.err()` with the result.

`msg`(*event=None*, ***kw*)
Process event and call `log.msg()` with the result.

`new`(***new_values*)
Clear context and binds *initial_values* using *`bind()`*.

Only necessary with dict implementations that keep global state like those wrapped by *`structlog.threadlocal.wrap_dict()`* when threads are re-used.

Return type *self.__class__*

`unbind`(**keys*)
Return a new logger with *keys* removed from the context.

Raises KeyError If the key is not part of the context.

Return type *self.__class__*

class `structlog.twisted.`**`LoggerFactory`**
Build a Twisted logger when an *instance* is called.

```
>>> from structlog import configure
>>> from structlog.twisted import LoggerFactory
>>> configure(logger_factory=LoggerFactory())
```

`__call__`(**args*)
Positional arguments are silently ignored.

Rvalue A new Twisted logger.

Changed in version 0.4.0: Added support for optional positional arguments.

class `structlog.twisted.`**`EventAdapter`**(*dictRenderer=None*)
Adapt an `event_dict` to Twisted logging system.

Particularly, make a wrapped twisted.python.log.err behave as expected.

Parameters **`dictRenderer`** (*callable*) – Renderer that is used for the actual log message. Please note that structlog comes with a dedicated *`JSONRenderer`*.

Must be the last processor in the chain and requires a *dictRenderer* for the actual formatting as an constructor argument in order to be able to fully support the original behaviors of `log.msg()` and `log.err()`.

class `structlog.twisted.`**`JSONRenderer`**(***dumps_kw*)
Behaves like *`structlog.processors.JSONRenderer`* except that it formats tracebacks and failures itself if called with *err()*.

Note: This ultimately means that the messages get logged out using *msg()*, and *not err()* which renders failures in separate lines.

Therefore it will break your tests that contain assertions using flushLoggedErrors.

Not an adapter like *`EventAdapter`* but a real formatter. Nor does it require to be adapted using it.

Use together with a *`JSONLogObserverWrapper`*-wrapped Twisted logger like *`plainJSONStdOutLogger()`* for pure-JSON logs.

`structlog.twisted.`**`plainJSONStdOutLogger`**()
Return a logger that writes only the message to stdout.

Transforms non-*`JSONRenderer`* messages to JSON.

Ideal for JSONifying log entries from Twisted plugins and libraries that are outside of your control:

```
$ twistd -n --logger structlog.twisted.plainJSONStdOutLogger web
{"event": "Log opened.", "system": "-"}
{"event": "twistd 13.1.0 (python 2.7.3) starting up.", "system": "-"}
{"event": "reactor class: twisted...EPollReactor.", "system": "-"}
{"event": "Site starting on 8080", "system": "-"}
{"event": "Starting factory <twisted.web.server.Site ...>", ...}
...
```

Composes *`PlainFileLogObserver`* and *`JSONLogObserverWrapper`* to a usable logger.

New in version 0.2.0.

`structlog.twisted.`**`JSONLogObserverWrapper`**(*observer*)
Wrap a log *observer* and render non-*`JSONRenderer`* entries to JSON.

Parameters **`observer`** (*ILogObserver*) – Twisted log observer to wrap. For example `PlainFileObserver` or Twisted's stock FileLogObserver

New in version 0.2.0.

class `structlog.twisted.`**`PlainFileLogObserver`**(*file*)
Write only the the plain message without timestamps or anything else.

Great to just print JSON to stdout where you catch it with something like runit.

Parameters **`file`** (*file*) – File to print to.

New in version 0.2.0.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

P

R

S

T

U

W