structlog Documentation

Release

Author

September 12, 2013

CONTENTS

Release v0.1.0 (*Installation*).

structlog makes structured logging in Python easy by *augmenting* your *existing* logger. It's licensed under the permissive Apache License, version 2, available from PyPI, and the source code can be found on GitHub. The full documentation is on Read the Docs.

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

CHAPTER

ONE

WHY YOU WANT 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 programer 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.

CHAPTER

TWO

WHY YOU WANT TO USE 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):

```
>>> log = log.bind(user='anonymous', some_key=23)
>>> log = log.bind(user='hynek', source='http', another_key=42)
>>> log.info('user.logged_in', happy=True)
some_key=23 user='hynek' source='http' 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.

CHAPTER

THREE

WHY YOU CAN START USING STRUCTLOG TODAY

- You can use both your bare logger and as well as the same logger wrapped by structlog at the same time. structlog avoids monkeypatching so a peaceful co-existence between various loggers is unproblematic.
- Events are free-form and interpreted as strings by default. Therefore the transition from traditional to structured logging is seamless most of the time. Just start wrapping your logger of choice and bind values later.
- If you don't like the idea of keeping the context within a local logger instance like in the example above, structlog offers transparent *thread local* storage for your context.

Intrigued? *Get started now* or have a look at more realistic *examples* and be completely convinced!

CHAPTER

FOUR

USER'S GUIDE

4.1 Getting Started

4.1.1 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.

4.1.2 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.

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.

4.1.3 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')
    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.

4.1.4 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.

4.1.5 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*!

4.2 Loggers

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

All it does is:

- Keeping a *context dictionary* and a *logger* that it's wrapping,
- recreating itself with (optional) *additional* context data (the `bind()` and `new()` methods),
- recreating itself with *less* data (`unbind()`),
- and finally relaying *all* other method calls to the wrapped logger 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)
>>> 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 an 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* how this may look like.

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.

4.2.1 Shipped Loggers

To save you the hassle of using standard library logging for simple stdout logging, structlog ships a `PrintLogger`. 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
```

4.2.2 Configuration

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 previous example 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. Passed *processors*, *wrapper_class*, and *context_class* arguments to `structlog.wrap_logger()` *always* take the highest precedence. 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 `[format_exc_info(), KeyValueRenderer]` for the processor chain.

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.

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.

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!'
```

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.

4.2.3 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[1]. Sooner or later, global state and mutable data lead to unpleasant surprises.

However, in the case of conventional web development, I 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.

4.2.4 Thread Local Context

Thread local storage makes your logger's context global but *only within the current thread*[2]. 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:

```
>>> 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())
```

[1] 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.

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

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:

```
>>> 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 I 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.

4.3 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.

4.3.1 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 BoundLogger, PrintLogger
wrapped_logger = PrintLogger()
logger = BoundLogger.wrap(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`.

4.3.2 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
```

4.3.3 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*.

4.4 Examples

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

4.4.1 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:
    # request_id='ffcdc44f-b952-4b5f-95e6-0f1f3a9ee5fd' event='user logged in' user='test-user'
    # ...
    some_function()
    # ...

if __name__ == "__main__":
    from structlog.stdlib import LoggerFactory
    from structlog.threadlocal import wrap_dict
    structlog.configure(
        context_class=wrap_dict(dict),
        logger_factory=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:
    # request_id='ffcdc44f-b952-4b5f-95e6-0f1f3a9ee5fd' something='shot_in_foot' event='user did some
```

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.

4.4.2 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.get_logger()


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__":
    from structlog.twisted import LoggerFactory, EventAdapter
    structlog.configure(
```

```
        processors=[EventAdapter()],
        logger_factory=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.

4.4.3 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*.

4.4.4 Custom Wrapper Classes

A custom wrapper class helps you to cast the shackles of your underlying logging system even further and get rid of even more boilerplate.

```
>>> from structlog import BoundLogger, PrintLogger, wrap_logger
>>> class SemanticLogger(BoundLogger):
...     def msg(self, event, **kw):
...         if not 'status' in kw:
...             self.info(event, status='ok', **kw)
...         else:
...             self.info(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'
```

I like to have semantically meaningful logger names. If you agree, this is a nice way to achieve that.

Of course, you can *configure* default processors, the wrapper class and the context classes globally.

CHAPTER

FIVE

API

5.1 structlog Package

5.1.1 `structlog` Package

`structlog.`**`get_logger`**`(`*`**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 **initial_values** – Values that are used to pre-populate your contexts.

See *Configuration* for details.

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

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

CamelCase alias for `structlog.get_logger()`.

This function is supposed to be in every source file – I don't want it to stick out like a sore thumb in frameworks like Twisted or Zope.

`structlog.`**`wrap_logger`**`(`*`logger`*`, `*`processors=None`*`, `*`wrapper_class=None`*`, `*`context_class=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 *processors* or *context_class* here, calls to `configure()` have *no* effect for the *respective* attribute.

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

Parameters

- **logger** – An instance of a logger whose method calls will be wrapped. Use configured logger factory if *None*.
- **processors** (*list of callables*) – List of processors.
- **wrapper_class** (*type*) – Class to use for wrapping loggers instead of `structlog.BoundLogger`.
- **context_class** (*type*) – Class to be used for internal dictionary.

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

structlog.**configure**(*processors=None*, *wrapper_class=None*, *context_class=None*, *logger_factory=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`.
- **context_class** – Class to be used for internal dictionary.

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 `[format_exc_info(), KeyValueRenderer]` for *processors*, `BoundLogger` for *wrapper_class*, `OrderedDict` for *context_class*, and `PrintLogger` for *logger_factory*.

Also sets the global class attribute `is_configured` to *True*.

class structlog.**BoundLogger**(*logger*, *processors*, *context*)

Immutable, context-carrying wrapper.

Public only for sub-classing, not intended to be instantiated by yourself. See `wrap_logger()` and `get_logger()`.

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 `BoundLogger`

bind(***new_values*)

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

Return type `BoundLogger`

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 `BoundLogger`

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.

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

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

Useful for unit tests.

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

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

5.1.2 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.

5.1.3 processors Module

Processors useful regardless of the logging framework.

class structlog.processors.**JSONRenderer**(***dumps_kw*)
Bases: object

Render the *event_dict* using *json.dumps(event_dict, **json_kw)*.

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

class structlog.processors.**KeyValueRenderer**(*sort_keys=False*)
Bases: object

Render *event_dict* as a list of `Key=repr(Value)` pairs.

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

Parameters **sort_keys** (*bool*) – Whether to sort keys when formatting.

class structlog.processors.**TimeStamper**(*fmt=None*, *utc=True*)
Bases: object

Add a timestamp to *event_dict*.

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'}
```

class structlog.processors.**UnicodeEncoder**(*encoding='utf-8'*, *errors='backslashreplace'*)
Bases: object

Encode unicode values in *event_dict*.

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'"
```

Just put it in the processor chain before *KeyValueRenderer*.

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):...
```

5.1.4 `stdlib` Module

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

class `structlog.stdlib.`**`LoggerFactory`**
Build a standard library logger when an *instance* is called.

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

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

Return type *logging.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
```

5.1.5 `twisted` Module

Processors and tools specific to the Twisted networking engine.

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())
```

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

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

Must be the last processor in the chain and requires a *dictFormatter* 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()*.

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

CHAPTER

SIX

ADDITIONAL NOTES

6.1 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.

6.1.1 Authors

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

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

- Alex Gaynor
- Christopher Armstrong
- Daniel Lindsley
- David Reid
- Donald Stufft
- Glyph
- Holger Krekel
- 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.

6.2 History

6.2.1 0.1.0 (2013-09-12)

- Initial release.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S