structlog Documentation

Release

Author

Jan 27, 2018

Contents

Release v18.1.0 (*What's new?*).

`structlog` makes logging in Python less painful and more powerful by adding structure to your log entries.

It's up to you whether you want `structlog` to take care about the **output** of your log entries or whether you prefer to **forward** them to an existing logging system like the standard library's `logging` module.

First steps:

- If you're not sure whether `structlog` is for you, have a look at *Why...*.
- If you can't wait to log your first entry, start at *Getting Started* and then work yourself through the basic docs.
- Once you have basic grasp of how `structlog` works, acquaint yourself with the *integrations* structlog is shipping with.

CHAPTER 1

User's Guide

1.1 Basics

1.1.1 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?

Easier Logging

You can stop writing prose and start thinking in terms of an event that happens in the context of key/value pairs:

```
>>> from structlog import get_logger
>>> log = get_logger()
>>> log.info("key_value_logging", out_of_the_box=True, effort=0)
2016-04-20 16:20.13 key_value_logging             effort=0 out_of_the_box=True
```

Each log entry is a meaningful dictionary instead of an opaque string now!

Data Binding

Since log entries are dictionaries, you can start binding and re-binding key/value pairs to your loggers to ensure they are present in every following logging call:

```
>>> log = log.bind(user="anonymous", some_key=23)
>>> log = log.bind(user="hynek", another_key=42)
>>> log.info("user.logged_in", happy=True)
2016-04-20 16:20.13 user.logged_in                 another_key=42 happy=True some_
↪key=23 user='hynek'
```

Powerful Pipelines

Each log entry goes through a processor pipeline that is just a chain of functions that receive a dictionary and return a new dictionary that gets fed into the next function. That allows for simple but powerful data manipulation:

```
def timestamper(logger, log_method, event_dict):
    """Add a timestamp to each log entry."""
    event_dict["timestamp"] = time.time()
    return event_dict
```

There are plenty of processors for most common tasks coming with `structlog`:

- Collectors of call stack information ("How did this log entry happen?"),
- ... and exceptions ("What happened?!").
- Unicode encoders/decoders.
- Flexible timestamping.

Formatting

`structlog` is completely flexible about *how* the resulting log entry is emitted. Since each log entry is a dictionary, it can be formatted to **any** format:

- A colorful key/value format for local development,
- JSON for easy parsing,
- or some standard format you have parsers for like nginx or Apache httpd.

Internally, formatters are processors whose return value (usually a string) is passed into loggers that are responsible for the output of your message. `structlog` comes with multiple useful formatters out of-the-box.

Output

`structlog` is also very flexible with the final output of your log entries:

- A **built-in** lightweight printer like in the examples above. Easy to use and fast.
- Use the **standard library**'s or **Twisted**'s logging modules for compatibility. In this case `structlog` works like a wrapper that formats a string and passes them off into existing systems that won't ever know that `structlog` even exists. Or the other way round: `structlog` comes with a `logging` formatter that allows for processing third party log records.

- Don't format it to a string at all! `structlog` passes you a dictionary and you can do with it whatever you want. Reported uses cases are sending them out via network or saving them in a database.

1.1.2 Getting Started

Installation

`structlog` can be easily installed using:

```
$ pip install structlog
```

If you'd like colorful output in development (you know you do!), install using:

```
$ pip install structlog colorama
```

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])
2016-09-17 10:13.45 greeted                       more_than_a_string=[1, 2, 3] whom=
↪'world'
```

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 or doing nothing.
- All keywords are formatted using *`structlog.dev.ConsoleRenderer`*. That in turn uses repr() to serialize all values to strings. Thus, it's easy to add support for logging of your own objects*[0].
- If you have colorama installed, it's rendered in nice *colors*.

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

Note: For brewity and to enable doctests, all further examples in `structlog`'s documentation use the more simplistic *`structlog.processors.KeyValueRenderer()`* without timestamps.

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:

[0] 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.

```
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.

- If you don't like the concept of pre-building a context: just don't! Convenient key-value-based logging is great to have on its own.
- To keep as much order of the keys as possible, an `collections.OrderedDict` is used for the context by default for Pythons that do not have ordered dictionaries by default (notably all versions of CPython before 3.6).
- 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.

Manipulating Log Entries in Flight

Now that your log events are dictionaries, it's also much easier to manipulate them than if it were plain strings.

To fascilitate that, `structlog` has the concept of *processor chains*. A processor is a callable like a function that receives the event dictionary along with two other arguments and returns a new event dictionary that may or may not differ from the one it got passed. The next processor in the chain receives that returned dictionary instead of the original one.

Let's assume you wanted to add a timestamp to every event dict. The processor would look like this:

```
>>> import datetime
>>> def timestamper(_, __, event_dict):
...     event_dict["time"] = datetime.datetime.now().isoformat()
...     return event_dict
```

Plain Python, plain dictionaries. No you have to tell `structlog` about your processor by *configuring* it:

```
>>> structlog.configure(processors=[timestamper, structlog.processors.
↪KeyValueRenderer()])
>>> structlog.get_logger().msg("hi")
event='hi' time='2018-01-21T09:37:36.976816'
```

Rendering

Finally you want to have control over the actual format of your log entries.

As you may have noticed in the previous section, renderers are just processors too. It's also important to note, that they do not necessarily have to render your event dictionary to a string. It depends on the *logger* that is wrapped by `structlog` what kind of input it should get.

However, in most cases it's gonna be strings.

So assuming you want to follow *best practices* and render your event dictionary to JSON that is picked up by a log aggregation system like ELK or Graylog, `structlog` comes with batteries included – you just have to tell it to use its *`JSONRenderer`*:

```
>>> structlog.configure(processors=[structlog.processors.JSONRenderer()])
>>> structlog.get_logger().msg("hi")
{"event": "hi"}
```

`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.stdlib import LoggerFactory
>>> structlog.configure(logger_factory=LoggerFactory())
>>> log = structlog.get_logger()
>>> log.warning("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*!

1.1.3 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*[0] 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")
```

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

```
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*.

On Python versions that have ordered dictionaries (Python 3.6+, PyPy) the default is a plain `dict`. For everything else it's `collections.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 and slowdown of using standard library's `logging` for 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!
```

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'})
```

1.1.4 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 camelCase-friendly alias *`structlog.getLogger()`* for y'all Twisted and Zope aficionados).

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

```
from structlog 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. 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*).

You can call *`structlog.configure()`* repeatedly and only set one or more settings – the rest will not be affected.

`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 try to be convenient and development-friendly.

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

At any time, you can check whether and how structlog is configured:

```
>>> structlog.is_configured()
False
>>> class MyDict(dict): pass
>>> structlog.configure(context_class=MyDict)
>>> structlog.is_configured()
True
>>> cfg = structlog.get_config()
>>> cfg["context_class"]
<class 'MyDict'>
```

Note: Since you will call *structlog.get_logger()* most likely in module scope, they run at import time before you had a chance to configure structlog. Hence they return a **lazy proxy** that returns a correct wrapped logger on first bind()/new().

Therefore, you must never call new() or bind() in module or class scope becauce otherwise you will receive a logger configured with structlog's default values. 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*. We 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.

1.1.5 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*[0]. 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*

[0] 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.

`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*†[0]. 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, 'b': 2, 'c': 3})>
```

To enable thread local context use the generated class as the context class:

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

Note: Creation of a new `BoundLogger` initializes the logger's context as `context_class(initial_values)`, and then adds any values passed via `.bind()`. As all instances of a wrapped dict-like class share the same data, in the case above, the new logger's context will contain all previously bound values in addition to the new ones.

`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()`*:

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

```
>>> 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!")
x=23 y='foo' 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.

1.1.6 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.warning("foo")`, it will be `"warning"`.

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
```

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.

It can return one of three types:

- A string that is passed as the first (and only) positional argument to the underlying logger.
- A tuple of `(args, kwargs)` that are passed as `log_method(*args, **kwargs)`.
- A dictionary which is passed as `log_method(**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.

Changed in version 14.0.0: Allow final processor to return a *dict*.

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*.

Third Party Packages

Since processors are self-contained callables, it's easy to write your own and to share them in separate packages. The following processor packages are known to be currently available on PyPI:

- structlog-pretty: Processors for prettier output – a code syntax highlighter, JSON and XML prettifiers, a multi-line string printer, and a numeric value rounder.

Please feel free to submit a pull request to extend this list with *your* package!

1.1.7 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 logging
import sys
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()
    # ...
    return "logged in!"

if __name__ == "__main__":
    logging.basicConfig(
        format="%(message)s",
        stream=sys.stdout,
        level=logging.INFO,
    )
    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_in_foot'
```

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
```

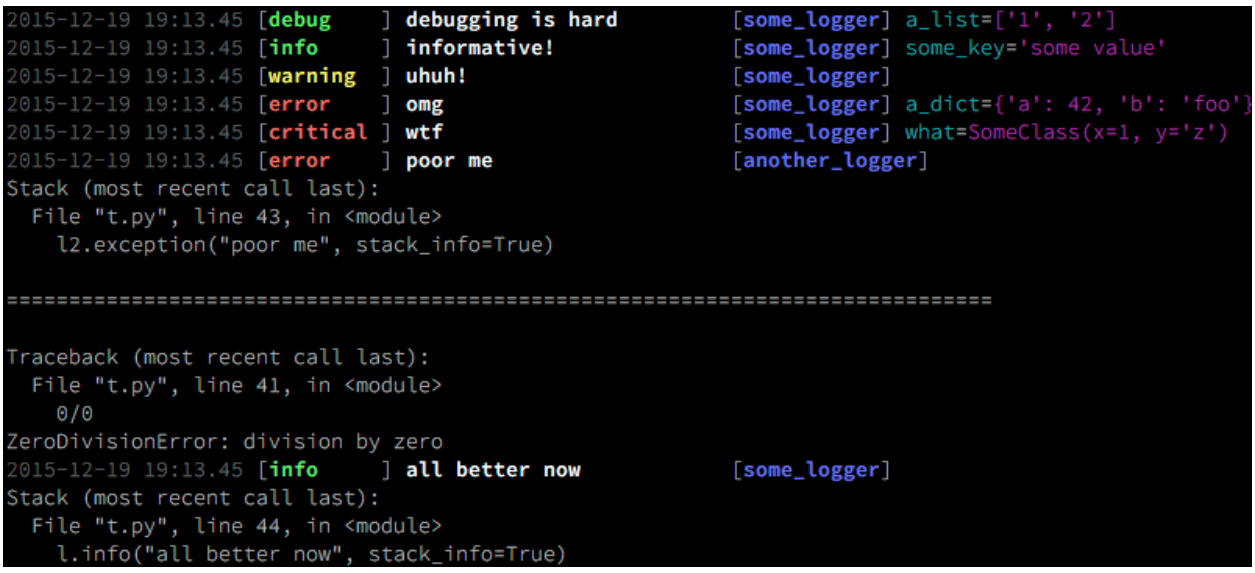
```
>>> from structlog.stdlib import filter_by_level
>>> logging.basicConfig(stream=sys.stdout, format="%(message)s")
>>> def add_timestamp(_, __, event_dict):
...     event_dict["timestamp"] = datetime.datetime.utcnow()
...     return event_dict
>>> def censor_password(_, __, event_dict):
...     pw = 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*.

1.1.8 Development

To make development a more pleasurable experience, `structlog` comes with the *`structlog.dev`* module.

The highlight is *`structlog.dev.ConsoleRenderer`* that offers nicely aligned and colorful console output while in development:



To use it, just add it as a renderer to your processor chain. It will recognize logger names, log levels, time stamps, stack infos, and tracebacks as produced by `structlog`'s processors and render them in special ways.

`structlog`'s default configuration already uses it, but if you want to use it along with standard library logging, we suggest the following configuration:

```
import structlog

structlog.configure(
    processors=[
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.TimeStamper(fmt="%Y-%m-%d %H:%M.%S"),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.dev.ConsoleRenderer()  # <===
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    wrapper_class=structlog.stdlib.BoundLogger,
    cache_logger_on_first_use=True,
)
```

1.2 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.

1.2.1 Standard Library Logging

Ideally, `structlog` should be able to be used as a drop-in replacement for standard library's `logging` by wrapping it. In other words, you should be able to replace your call to `logging.getLogger()` by a call to *`structlog.get_logger()`* and things should keep working as before (if `structlog` is configured right, see *Suggested Configurations* below).

If you run into incompatibilities, it is a *bug* so please take the time to report it! If you're a heavy `logging` user, your help to ensure a better compatibility would be highly appreciated!

Just Enough `logging`

If you want to use `structlog` with `logging`, you still have to have at least fleeting understanding on how the standard library operates because `structlog` will *not* do any magic things in the background for you. Most importantly you have *configure* the `logging` system *additionally* to configuring `structlog`.

Usually it is enough to use:

```
import logging
import sys

logging.basicConfig(
    format="%(message)s",
    stream=sys.stdout,
```

```
    level=logging.INFO,
)
```

This will send all log messages with the log level `logging.INFO` and above (that means that e.g. `logging.debug()` calls are ignored) to standard out without any special formatting by the standard library.

If you require more complex behavior, please refer to the standard library's `logging` documentation.

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 `logging.Logger`,
- hence causing less cryptic error messages if you get method names wrong.

Processors

`structlog` comes with a few standard library-specific processors:

***`render_to_log_kwargs()`*:** Renders the event dictionary into keyword arguments for `logging.log()` that attaches everything except the *event* field to the *extra* argument. This is useful if you want to render your log entries entirely within `logging`.

***`filter_by_level()`*:** Checks the log entry'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.

***`add_logger_name()`*:** Adds the name of the logger to the event dictionary under the key `logger`.

***`add_log_level()`*:** Adds the log level to the event dictionary under the key `level`.

***`PositionalArgumentsFormatter`*:** This processes and formats positional arguments (if any) passed to log methods in the same way the `logging` module would do, e.g. `logger.info("Hello, %s", name)`.

`structlog` also comes with *`ProcessorFormatter`* which is a `logging.Formatter` that enables you to format non-`structlog` log entries using `structlog` renderers *and* multiplex `structlog`'s output with different renderers (see below for an example).

Suggested Configurations

Depending *where* you'd like to do your formatting, you can take one of three approaches:

Rendering Using `logging`-based Formatters

```
import structlog

structlog.configure(
    processors=[
        structlog.stdlib.filter_by_level,
        structlog.stdlib.add_logger_name,
```

```
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.stdlib.render_to_log_kwargs,
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    wrapper_class=structlog.stdlib.BoundLogger,
    cache_logger_on_first_use=True,
)
```

Now you have the event dict available within each log record. If you want all your log entries (i.e. also those not from your app/`structlog`) to be formatted as JSON, you can use the python-json-logger library:

```
import logging
import sys

from pythonjsonlogger import jsonlogger

handler = logging.StreamHandler(sys.stdout)
handler.setFormatter(jsonlogger.JsonFormatter())
root_logger = logging.getLogger()
root_logger.addHandler(handler)
```

Now both `structlog` and `logging` will emit JSON logs:

```
>>> structlog.get_logger("test").warning("hello")
{"message": "hello", "logger": "test", "level": "warning"}

>>> logging.getLogger("test").warning("hello")
{"message": "hello"}
```

Rendering Using `structlog`-based Formatters Within `logging`

`structlog` comes with a *`ProcessorFormatter`* that can be used as a `Formatter` in any stdlib `Handler` object.

The *`ProcessorFormatter`* has two parts to its API:

1. The *`wrap_for_formatter()`* method must be used as the last processor in *`structlog.configure()`*, it converts the the processed event dict to something that the `ProcessorFormatter` understands.
2. The *`ProcessorFormatter`* itself, which can wrap any `structlog` renderer to handle the output of both `structlog` and standard library events.

Thus, the simplest possible configuration looks like the following:

```
import logging
import structlog

structlog.configure(
    processors=[
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    logger_factory=structlog.stdlib.LoggerFactory(),
```

```
)

formatter = structlog.stdlib.ProcessorFormatter(
    processor=structlog.dev.ConsoleRenderer(),
)

handler = logging.StreamHandler()
handler.setFormatter(formatter)
root_logger = logging.getLogger()
root_logger.addHandler(handler)
root_logger.setLevel(logging.INFO)
```

which will allow both of these to work in other modules:

```
>>> import logging
>>> import structlog

>>> logging.getLogger("stdlog").info("woo")
woo
>>> structlog.get_logger("structlog").info("amazing", events="oh yes")
amazing                        events=oh yes
```

Of course, you probably want timestamps and log levels in your output. The *ProcessorFormatter* has a `foreign_pre_chain` argument which is responsible for adding properties to events from the standard library – i.e. that do not originate from a `structlog` logger – and which should in general match the `processors` argument to *structlog.configure()* so you get a consistent output.

For example, to add timestamps, log levels, and traceback handling to your logs you should do:

```
timestamper = structlog.processors.TimeStamper(fmt="%Y-%m-%d %H:%M:%S")
shared_processors = [
    structlog.stdlib.add_log_level,
    timestamper,
]

structlog.configure(
    processors=shared_processors + [
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    logger_factory=structlog.stdlib.LoggerFactory(),
    cache_logger_on_first_use=True,
)

formatter = structlog.stdlib.ProcessorFormatter(
    processor=structlog.dev.ConsoleRenderer(),
    foreign_pre_chain=shared_processors,
)
```

which (given the same `logging.*` calls as in the previous example) will result in:

```
>>> logging.getLogger("stdlog").info("woo")
2017-03-06 14:59:20 [info     ] woo
>>> structlog.get_logger("structlog").info("amazing", events="oh yes")
2017-03-06 14:59:20 [info     ] amazing                        events=oh yes
```

This allows you to set up some sophisticated logging configurations. For example, to use the standard library's `dictConfig()` to log colored logs to the console and plain logs to a file you could do:

```
import logging.config
import structlog

timestamper = structlog.processors.TimeStamper(fmt="%Y-%m-%d %H:%M:%S")
pre_chain = [
    # Add the log level and a timestamp to the event_dict if the log entry
    # is not from structlog.
    structlog.stdlib.add_log_level,
    timestamper,
]

logging.config.dictConfig({
        "version": 1,
        "disable_existing_loggers": False,
        "formatters": {
            "plain": {
                "()": structlog.stdlib.ProcessorFormatter,
                "processor": structlog.dev.ConsoleRenderer(colors=False),
                "foreign_pre_chain": pre_chain,
            },
            "colored": {
                "()": structlog.stdlib.ProcessorFormatter,
                "processor": structlog.dev.ConsoleRenderer(colors=True),
                "foreign_pre_chain": pre_chain,
            },
        },
        "handlers": {
            "default": {
                "level": "DEBUG",
                "class": "logging.StreamHandler",
                "formatter": "colored",
            },
            "file": {
                "level": "DEBUG",
                "class": "logging.handlers.WatchedFileHandler",
                "filename": "test.log",
                "formatter": "plain",
            },
        },
        "loggers": {
            "": {
                "handlers": ["default", "file"],
                "level": "DEBUG",
                "propagate": True,
            },
        }
})
structlog.configure(
    processors=[
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        timestamper,
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
```

```
    wrapper_class=structlog.stdlib.BoundLogger,
    cache_logger_on_first_use=True,
)
```

This defines two formatters: one plain and one colored. Both are run for each log entry. Log entries that do not originate from `structlog`, are additionally pre-processed using a cached `timestamper` and *`add_log_level()`*.

```
>>> logging.getLogger().warning("bar")
2017-03-06 11:49:27 [warning  ] bar

>>> structlog.get_logger("structlog").warning("foo", x=42)
2017-03-06 11:49:32 [warning  ] foo                              x=42

>>> print(open("test.log").read())
2017-03-06 11:49:27 [warning  ] bar
2017-03-06 11:49:32 [warning  ] foo                              x=42
```

(sadly, you have to imagine the colors in the first two outputs)

If you leave `foreign_pre_chain` *None*, formatting will be left to `logging`. Meaning: you can define a `format` for *`ProcessorFormatter`* too!

Rendering Within `structlog`

A basic configuration to output structured logs in JSON format looks like this:

```
import structlog

structlog.configure(
    processors=[
        structlog.stdlib.filter_by_level,
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.processors.JSONRenderer()
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    wrapper_class=structlog.stdlib.BoundLogger,
    cache_logger_on_first_use=True,
)
```

(if you're still runnning Python 2, replace *`UnicodeDecoder`* through *`UnicodeEncoder`*)

To make your program behave like a proper 12 factor app that outputs only JSON to `stdout`, configure the `logging` module like this:

```
import logging
import sys

logging.basicConfig(
    format="%(message)s",
    stream=sys.stdout,
```

```
    level=logging.INFO,
)
```

In this case *only* your own logs are formatted as JSON:

```
>>> structlog.get_logger("test").warning("hello")
{"event": "hello", "logger": "test", "level": "warning", "timestamp": "2017-03-
↪06T07:39:09.518720Z"}

>>> logging.getLogger("test").warning("hello")
hello
```

1.2.2 Twisted

Warning: Since `sys.exc_clear()` has been dropped in Python 3, there is currently no way to avoid multiple tracebacks in your log files if using `structlog` together with Twisted on Python 3.

Note: `structlog` currently only supports the legacy – but still perfectly working – Twisted logging system found in `twisted.python.log`.

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*.

1.2.3 Logging Best Practices

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.

Therefore let's rely on proven tools as much as possible and do only the absolutely necessary inside of Python*[0].

A simple but powerful approach is to log to unbuffered standard out and let other tools take care of the rest. That can be your terminal window while developing, it can be systemd redirecting your log entries it to syslogd, or your cluster manager. It doesn't matter where or how your application is running, it just works.

This is why the popular twelve-factor app methodology suggests just that.

Centralized Logging

Nowadays you usually don't want your logfiles in compressed archives distributed over dozens – if not thousands – servers or cluster nodes. You want them in a single location; parsed, indexed and easy to search.

ELK

The ELK stack (Elasticsearch, Logstash, Kibana) from Elastic is a great way to store, parse, and search your logs.

The way it works is that you have local log shippers like Filebeat that parse your log files and forward the log entries to your Logstash server. Logstash parses the log entries and stores them in in Elasticsearch. Finally, you can view and search them in Kibana.

If your log entries consist of a JSON dictionary, this is fairly easy and efficient. All you have to do is to tell Logstash the name of you timestamp field.

Graylog

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`.

1.3 Advanced Topics

1.3.1 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).

[0] 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.

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.

1.3.2 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 predictable 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 *Standard Library Logging* 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 that later calls on *configure()* don't have any effect on already cached loggers – that shouldn't matter outside of testing though.

5. Use a faster JSON serializer than the standard library. Possible alternatives are among others simplejson or RapidJSON (Python 3 only):

```
structlog.processors.JSONRenderer(serializer=rapidjson.dumps)
```

CHAPTER 2

API Reference

2.1 API Reference

Note: The examples here use a very simplified configuration using the minimalistic *structlog.processors.KeyValueRenderer* for brewity and to enable doctests. The output is going to be different (nicer!) with default configuration.

2.1.1 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*)
CamelCase alias for *structlog.get_logger()*.

This function is supposed to be in every source file – we 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*, *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.

Can be called several times, keeping an argument at *None* leaves is unchanged from the current setting.

After calling for the first time, *is_configured()* starts returning `True`.

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 *Standard Library Logging*, *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 builtin defaults.

is_configured() starts returning `False` afterwards.

structlog.**is_configured**()
Return whether `structlog` has been configured.

If `False`, `structlog` is running with builtin defaults.

Return type bool

structlog.**get_config**()
Get a dictionary with the current configuration.

Note: Changes to the returned dictionary do *not* affect `structlog`.

Return type dict

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.stdlib.BoundLogger* or *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*)
Print events into a file.

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

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

Useful if you follow *current logging best practices*.

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

critical(*message*)
Print *message*.

debug(*message*)
Print *message*.

err(*message*)
Print *message*.

error(*message*)
Print *message*.

failure(*message*)
Print *message*.

info(*message*)
Print *message*.

log(*message*)
Print *message*.

msg(*message*)
Print *message*.

warning(*message*)
Print *message*.

class structlog.**PrintLoggerFactory**(*file=None*)
Produce *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**
Return the arguments that it's called with.

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

Useful for testing.

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

failure(**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**
Produce and cache *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*.

_logger = **None**
Wrapped logger.

Note: Despite underscore available **read-only** to custom wrapper classes.

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.

Raises `ValueError` if the final processor doesn't return a string, tuple, or a dict.

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

Note: Despite underscore available to custom wrapper classes.

See also *Custom Wrappers*.

Changed in version 14.0.0: Allow final processor to return a *dict*.

_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__*

2.1.2 dev Module

Helpers that make development with `structlog` more pleasant.

class `structlog.dev.`**ConsoleRenderer**(*pad_event=30*, *colors=True*, *force_colors=False*, *repr_native_str=False*, *level_styles=None*)
Render *event_dict* nicely aligned, possibly in colors, and ordered.

Parameters

- **pad_event** (*int*) – Pad the event to this many characters.
- **colors** (*bool*) – Use colors for a nicer output.

- **force_colors** (*bool*) – Force colors even for non-tty destinations. Use this option if your logs are stored in a file that is meant to be streamed to the console.
- **repr_native_str** (*bool*) – When `True`, `repr()` is also applied to native strings (i.e. unicode on Python 3 and bytes on Python 2). Setting this to `False` is useful if you want to have human-readable non-ASCII output on Python 2. The *event* key is *never* `repr()` -ed.
- **level_styles** (*dict*) – When present, use these styles for colors. This must be a dict from level names (strings) to colorama styles. The default can be obtained by calling *ConsoleRenderer.get_default_level_styles()*

Requires the colorama package if *colors* is `True`.

New in version 16.0.

New in version 16.1: *colors*

New in version 17.1: *repr_native_str*

New in version 18.1: *force_colors*

New in version 18.1: *level_styles*

static get_default_level_styles(*colors=True*)

Get the default styles for log levels

This is intended to be used with *ConsoleRenderer*'s `level_styles` parameter. For example, if you are adding custom levels in your home-grown *add_log_level()* you could do:

```
my_styles = ConsoleRenderer.get_default_level_styles()
my_styles["EVERYTHING_IS_ON_FIRE"] = my_styles["critical"]
renderer = ConsoleRenderer(level_styles=my_styles)
```

Parameters **colors** (*bool*) – Whether to use colorful styles. This must match the *colors* parameter to *ConsoleRenderer*. Default: True.

2.1.3 `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")
x=5 y=3 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** (structlog.BoundLogger) – A logger with *possibly* thread local state.

Return type *BoundLogger* with an immutable context.

2.1.4 processors Module

Processors useful regardless of the logging framework.

class structlog.processors.**JSONRenderer**(*serializer=<function dumps>*, ***dumps_kw*)
Render the *event_dict* using `serializer(event_dict, **json_kw)`.

Parameters

- **json_kw** (*dict*) – Are passed unmodified to *serializer*.
- **serializer** (*callable*) – A `json.dumps()`-compatible callable that will be used to format the string. This can be used to use alternative JSON encoders like simplejson or RapidJSON (faster but Python 3-only) (default: `json.dumps()`).

New in version 0.2.0: Support for `__structlog__` serialization method.

New in version 15.4.0: `serializer` parameter.

```
>>> 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.

class structlog.processors.**KeyValueRenderer**(*sort_keys=False*, *key_order=None*, *drop_missing=False*, *repr_native_str=True*)
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.
- **drop_missing** (*bool*) – When `True`, extra keys in *key_order* will be dropped rather than rendered as `None`.

- **repr_native_str** (*bool*) – When `True`, `repr()` is also applied to native strings (i.e. unicode on Python 3 and bytes on Python 2). Setting this to `False` is useful if you want to have human-readable non-ASCII output on Python 2.

New in version 0.2.0: *key_order*

New in version 16.1.0: *drop_missing*

New in version 17.1.0: *repr_native_str*

```
>>> 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'
```

class `structlog.processors.`**UnicodeDecoder**(*encoding='utf-8'*, *errors='replace'*)
Decode byte string values in *event_dict*.

Parameters

- **encoding** (*str*) – Encoding to decode from (default: `"utf-8"`).
- **errors** (*str*) – How to cope with encoding errors (default: `"replace"`).

Useful if you're running Python 3 as otherwise `b"abc"` will be rendered as `'b"abc"'`.

Just put it in the processor chain before the renderer.

New in version 15.4.0.

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 if you're running Python 2 as otherwise `u"abc"` will be rendered as `'u"abc"'`.

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 is an Exception *and* you're running Python 3, 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.

Changed in version 16.0.0: Added support for passing exceptions as exc_info on Python 3.

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

Note: You should let OS tools take care of timestamping. See also *Logging Best Practices*.

Parameters

- **fmt** (*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.
- **key** (*str*) – Target key in *event_dict* for added timestamps.

```
>>> 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", key="year")(None, None, {})
{'year': '2013'}
```

2.1.5 stdlib Module

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

See also *structlog's standard library support*.

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:

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

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

Return type *self.__class__*

critical(*event=None*, **args*, ***kw*)
Process event and call `logging.Logger.critical()` with the result.

debug(*event=None*, **args*, ***kw*)
Process event and call `logging.Logger.debug()` with the result.

error(*event=None*, **args*, ***kw*)
Process event and call `logging.Logger.error()` with the result.

exception(*event=None*, **args*, ***kw*)
Process event and call `logging.Logger.error()` with the result, after setting `exc_info` to *True*.

info(*event=None*, **args*, ***kw*)
Process event and call `logging.Logger.info()` with the result.

log(*level*, *event*, **args*, ***kw*)
Process event and call the appropriate logging method depending on *level*.

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__*

warn(*event=None*, **args*, ***kw*)
Process event and call `logging.Logger.warning()` with the result.

warning(*event=None*, **args*, ***kw*)
Process event and call `logging.Logger.warning()` with the result.

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

Sets a custom logger using `logging.setLoggerClass()` 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 `structlog.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.`**render_to_log_kwargs**(*wrapped_logger*, *method_name*, *event_dict*)
Render *event_dict* into keyword arguments for `logging.log()`.

The *event* field is translated into *msg* and the rest of the *event_dict* is added as *extra*.

This allows you to defer formatting to `logging`.

New in version 17.1.0.

`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
```

`structlog.stdlib.`**add_log_level**(*logger*, *method_name*, *event_dict*)
Add the log level to the event dict.

`structlog.stdlib.`**add_logger_name**(*logger*, *method_name*, *event_dict*)
Add the logger name to the event dict.

class `structlog.stdlib.`**PositionalArgumentsFormatter**(*remove_positional_args=True*)
Apply stdlib-like string formatting to the *event* key.

If the *positional_args* key in the event dict is set, it must contain a tuple that is used for formatting (using the *%s* string formatting operator) of the value from the *event* key. This works in the same way as the stdlib handles arguments to the various log methods: if the tuple contains only a single *dict* argument it is used for keyword placeholders in the *event* string, otherwise it will be used for positional placeholders.

positional_args is populated by *structlog.stdlib.BoundLogger* or can be set manually.

The *remove_positional_args* flag can be set to *False* to keep the *positional_args* key in the event dict; by default it will be removed from the event dict after formatting a message.

class `structlog.stdlib.`**ProcessorFormatter**(*processor*, *foreign_pre_chain=None*, *keep_exc_info=False*, *keep_stack_info=False*, **args*, ***kwargs*)
Call `structlog` processors on `logging.LogRecord`s.

This `logging.Formatter` allows to configure `logging` to call *processor* on `structlog`-borne log entries (origin is determined solely on the fact whether the `msg` field on the `logging.LogRecord` is a dict or not).

This allows for two interesting use cases:

1. You can format non-`structlog` log entries.
2. You can multiplex log records into multiple `logging.Handler`s.

Please refer to *Standard Library Logging* for examples.

Parameters

- **`processor`** (*callable*) – A `structlog` processor.
- **`foreign_pre_chain`** – If not *None*, it is used as an iterable of processors that is applied to non-`structlog` log entries before *processor*. If *None*, formatting is left to `logging`. (default: *None*)
- **`keep_exc_info`** (*bool*) – `exc_info` on `logging.LogRecord`s is added to the `event_dict` and removed afterwards. Set this to `True` to keep it on the `logging.LogRecord`. (default: False)
- **`keep_stack_info`** (*bool*) – Same as *keep_exc_info* except for Python 3's `stack_info`.

Return type str

New in version 17.1.0.

New in version 17.2.0: *keep_exc_info* and *keep_stack_info*

static `wrap_for_formatter`(*logger*, *name*, *event_dict*)
Wrap *logger*, *name*, and *event_dict*.

The result is later unpacked by *`ProcessorFormatter`* when formatting log entries.

Use this static method as the renderer (i.e. final processor) if you want to use *`ProcessorFormatter`* in your `logging` configuration.

2.1.6 `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**(*serializer=<function dumps>*, ***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 3

Project Information

`structlog` is dual-licensed under Apache License, version 2 and MIT, available from PyPI, the source code can be found on GitHub, the documentation at http://www.structlog.org/.

`structlog` targets Python 2.7, 3.4 and newer, and PyPy.

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

3.1 Backward Compatibility

`structlog` has a very strong backward compatibility policy that is inspired by the one of the Twisted framework.

Put simply, you shouldn't ever be afraid to upgrade `structlog` if you're using its public APIs. If there will ever be need to break compatibility, it will be announced in the *Changelog* and raise deprecation warning for a year before it's finally really broken.

> **Warning:** You cannot however rely on the default settings and the *`structlog.dev`* module. They may be adjusted in the future to provide a better experience when starting to use `structlog`. So please make sure to **always** properly configure your applications.

3.2 How To Contribute

First off, thank you for considering contributing to `structlog`! It's people like *you* who make it is such a great tool for everyone.

This document is mainly to help you to get started by codifying tribal knowledge and expectations and make it more accessible to everyone. But don't be afraid to open half-finished PRs and ask questions if something is unclear!

3.2.1 Workflow

- No contribution is too small! Please submit as many fixes for typos and grammar bloopers as you can!
- Try to limit each pull request to *one* change only.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation can't be merged.
- Make sure your changes pass our CI. You won't get any feedback until it's green unless you ask for it.
- Once you've addressed review feedback, make sure to bump the pull request with a short note, so we know you're done.
- Don't break backward compatibility.

3.2.2 Code

- Obey PEP 8 and PEP 257. We use the `"""`-on-separate-lines style for docstrings:

```
def func(x):
    """
    Do something.

    :param str x: A very important parameter.

    :rtype: str
    """
```

- If you add or change public APIs, tag the docstring using `.. versionadded:: 16.0.0 WHAT` or `.. versionchanged:: 17.1.0 WHAT`.
- Prefer double quotes (`"`) over single quotes (`'`) unless the string contains double quotes itself.

3.2.3 Tests

- Write your asserts as `expected == actual` to line them up nicely:

```
x = f()

assert 42 == x.some_attribute
assert "foo" == x._a_private_attribute
```

- To run the test suite, all you need is a recent tox. It will ensure the test suite runs with all dependencies against all Python versions just as it will on Travis CI. If you lack some Python versions, you can can make it a non-failure using `tox --skip-missing-interpreters` (in that case you may want to look into pyenv that makes it very easy to install many different Python versions in parallel).
- Write good test docstrings.

3.2.4 Documentation

- Use semantic newlines in reStructuredText files (files ending in `.rst`):

```
This is a sentence.
This is another sentence.
```

- If you start a new section, add two blank lines before and one blank line after the header except if two headers follow immediately after each other:

```
Last line of previous section.


Header of New Top Section
-------------------------

Header of New Section
^^^^^^^^^^^^^^^^^^^^^

First line of new section.
```

- If your change is noteworthy, add an entry to the changelog. Use semantic newlines, and add a link to your pull request:

```
- Added ``structlog.func()`` that does foo.
  It's pretty cool.
  [`#1 <https://github.com/hynek/structlog/pull/1>`_]
- ``structlog.func()`` now doesn't crash the Large Hadron Collider anymore.
  That was a nasty bug!
  [`#2 <https://github.com/hynek/structlog/pull/2>`_]
```

3.2.5 Local Development Environment

You can (and should) run our test suite using tox however you'll probably want a more traditional environment too. We highly recommend to develop using the latest Python 3 release because you're more likely to catch certain bugs earlier.

First create a virtual environment. It's out of scope for this document to list all the ways to manage virtual environments in Python but if you don't have already a pet way, take some time to look at tools like pew, virtualfish, and virtualenvwrapper.

Next get an up to date checkout of the `structlog` repository:

```
git checkout git@github.com:hynek/structlog.git
```

Change into the newly created directory and **after activating your virtual environment** install an editable version of `structlog` along with its test and docs dependencies:

```
cd structlog
pip install -e .[tests,docs]
```

If you run the virtual environment's Python and try to `import structlog` it should work!

At this point

```
python -m pytest
```

should work and pass

and

```
cd docs
make html
```

should build docs in `docs/_build/html`.

Again, this list is mainly to help you to get started by codifying tribal knowledge and expectations. If something is unclear, feel free to ask for help!

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms. Please report any harm to Hynek Schlawack in any way you find appropriate.

Thank you for considering contributing to `structlog`!

3.3 License and Hall of Fame

`structlog` is licensed both under the Apache License, Version 2 and the MIT license.

The reason for that is to be both protected against patent claims by own contributors and still allow the usage within GPLv2 software. For more legal details, see this issue on the bug tracker of PyCA's cryptography.

The full license texts can be also found in the source code repository:

- Apache License 2.0
- MIT

3.4 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.

A full list of contributors can be found on GitHub's overview. Some of them disapprove of the addition of thread local context data. :)

The `structlog` logo has been contributed by Russell Keith-Magee.

3.5 Changelog

Versions are year-based with a strict backward compatibility policy. The third digit is only for regressions.

3.5.1 18.1.0 (2018-01-27)

Backward-incompatible changes:

none

Deprecations:

- The meaning of the `structlog[dev]` installation target will change from "colorful output" to "dependencies to develop `structlog`" in 19.1.0.

  The main reason behind this decision is that it's impossible to have a `structlog` in your normal dependencies and additionally a `structlog[dev]` for developement (`pip` will report an error).

Changes:

- Empty strings are valid events now. #110
- Do not encapsulate Twisted failures twice with newer versions of Twisted. #144
- `structlog.dev.ConsoleRenderer` now accepts a *force_colors* argument to output colored logs even if the destination is not a tty. Use this option if your logs are stored in files that are intended to be streamed to the console.
- `structlog.dev.ConsoleRenderer` now accepts a *level_styles* argument for overriding the colors for individual levels, as well as to add new levels. See the docs for `ConsoleRenderer.get_default_level_styles()` for usage. #139
- `structlog.stdlib.BoundLogger.exception()` now uses the `exc_info` argument if it has been passed instead of setting it unconditionally to `True`. #149
- Default configuration now uses plain `dicts` on Python 3.6+ and PyPy since they are ordered by default.
- Added `structlog.is_configured()` to check whether or not `structlog` has been configured.
- Added `structlog.get_config()` to introspect current configuration.

3.5.2 17.2.0 (2017-05-15)

Backward-incompatible changes:

none

Deprecations:

none

Changes:

- `structlog.stdlib.ProcessorFormatter` now accepts *keep_exc_info* and *keep_stack_info* arguments to control what to do with this information on log records. Most likely you want them both to be `False` therefore it's the default. #109
- `structlog.stdlib.add_logger_name()` now works in `structlog.stdlib.ProcessorFormatter`'s `foreign_pre_chain`. #112
- Clear log record args in `structlog.stdlib.ProcessorFormatter` after rendering. This fix is for you if you tried to use it and got `TypeError: not all arguments converted during string formatting` exceptions. #116 #117

3.5.3 17.1.0 (2017-04-24)

The main features of this release are massive improvements in standard library's `logging` integration. Have a look at the updated standard library chapter on how to use them! Special thanks go to Fabian Büchler, Gilbert Gilb's, Iva Kaneva, insolite, and sky-code, that made them possible.

Backward-incompatible changes:

- The default renderer now is `structlog.dev.ConsoleRenderer` if you don't configure `structlog`. Colors are used if available and human-friendly timestamps are prepended. This is in line with our backward compatibility policy that explicitly excludes default settings.

Changes:

- Added `structlog.stdlib.render_to_log_kwargs()`. This allows you to use `logging`-based formatters to take care of rendering your entries. #98
- Added `structlog.stdlib.ProcessorFormatter` which does the opposite: This allows you to run `structlog` processors on arbitrary `logging.LogRecords`. #79 #105
- UNIX epoch timestamps from `structlog.processors.TimeStamper` are more precise now.
- Added *repr_native_str* to `structlog.processors.KeyValueRenderer` and `structlog.dev.ConsoleRenderer`. This allows for human-readable non-ASCII output on Python 2 (`repr()` on Python 2 behaves like `ascii()` on Python 3 in that regard). As per compatibility policy, it's on (original behavior) in `KeyValueRenderer` and off (humand-friendly behavior) in `ConsoleRenderer`. #94
- Added *colors* argument to `structlog.dev.ConsoleRenderer` and made it the default renderer. #78
- Fixed bug with Python 3 and `structlog.stdlib.BoundLogger.log()`. Error log level was not reproductible and was logged as exception one time out of two. #92
- Positional arguments are now removed even if they are empty. #82

3.5.4 16.1.0 (2016-05-24)

Backward-incompatible changes:

- Python 3.3 and 2.6 aren't supported anymore. They may work by chance but any effort to keep them working has ceased.

  The last Python 2.6 release was on October 29, 2013 and isn't supported by the CPython core team anymore. Major Python packages like Django and Twisted dropped Python 2.6 a while ago already.

  Python 3.3 never had a significant user base and wasn't part of any distribution's LTS release.

Changes:

- Add a `drop_missing` argument to `KeyValueRenderer`. If `key_order` is used and a key is missing a value, it's not rendered at all instead of being rendered as `None`. #67
- Exceptions without a `__traceback__` are now also rendered on Python 3.
- Don't cache loggers in lazy proxies returned from `get_logger()`. This lead to in-place mutation of them if used before configuration which in turn lead to the problem that configuration was applied only partially to them later. #72

3.5.5 16.0.0 (2016-01-28)

Changes:

- `structlog.processors.ExceptionPrettyPrinter` and `structlog.processors.format_exc_info` now support passing of Exceptions on Python 3.
- Clean up the context when exiting `structlog.threadlocal.tmp_bind` in case of exceptions. #64
- Be more more lenient about missing `__name__`s. #62
- Add `structlog.dev.ConsoleRenderer` that renders the event dictionary aligned and with colors.
- Use six for compatibility.
- Add `structlog.processors.UnicodeDecoder` that will decode all byte string values in an event dictionary to Unicode.
- Add `serializer` parameter to `structlog.processors.JSONRenderer` which allows for using different (possibly faster) JSON encoders than the standard library.

3.5.6 15.3.0 (2015-09-25)

Changes:

- Tolerate frames without a `__name__`, better. #58
- Officially support Python 3.5.
- Add `structlog.ReturnLogger.failure` and `structlog.PrintLogger.failure` as preparation for the new Twisted logging system.

3.5.7 15.2.0 (2015-06-10)

Changes:

- Allow empty lists of processors. This is a valid use case since #26 has been merged. Before, supplying an empty list resulted in the defaults being used.
- Prevent Twisted's `log.err` from quoting strings rendered by `structlog.twisted.JSONRenderer`.
- Better support of `logging.Logger.exception` within `structlog`. #52
- Add option to specify target key in `structlog.processors.TimeStamper` processor. #51

3.5.8 15.1.0 (2015-02-24)

Changes:

- Tolerate frames without a `__name__`.

3.5.9 15.0.0 (2015-01-23)

Changes:

- Add `structlog.stdlib.add_log_level` and `structlog.stdlib.add_logger_name` processors. #44
- Add `structlog.stdlib.BoundLogger.log`. #42
- Pass positional arguments to stdlib wrapped loggers that use string formatting. #19
- `structlog` is now dually licensed under the Apache License, Version 2 and the MIT license. Therefore it is now legal to use structlog with GPLv2-licensed projects. #28
- Add `structlog.stdlib.BoundLogger.exception`. #22

3.5.10 0.4.2 (2014-07-26)

Changes:

- 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. #8
- `structlog.PrintLogger` now is thread-safe.
- Test Twisted-related code on Python 3 (with some caveats).
- Drop support for Python 3.2. There is no justification to add complexity for a Python version that nobody uses. If you are one of the 0.350% that use Python 3.2, please stick to the 0.4 branch; critical bugs will still be fixed.
- Officially support Python 3.4.
- Allow final processor to return a dictionary. See the adapting chapter. #26
- `from structlog import *` works now (but you still shouldn't use it).

3.5.11 0.4.1 (2013-12-19)

Changes:

- Don't cache proxied methods in `structlog.threadlocal._ThreadLocalDictWrapper`. This doesn't affect regular users.
- Various doc fixes.

3.5.12 0.4.0 (2013-11-10)

Backward-incompatible changes:

Changes:

- Add `structlog.processors.StackInfoRenderer` for adding stack information to log entries without involving exceptions. Also added it to default processor chain. #6
- Allow optional positional arguments for `structlog.get_logger` that are passed to logger factories. The standard library factory uses this for explicit logger naming. #12
- Add `structlog.processors.ExceptionPrettyPrinter` for development and testing when multiline log entries aren't just acceptable but even helpful.
- Allow the standard library name guesser to ignore certain frame names. This is useful together with frameworks.
- Add meta data (e.g. function names, line numbers) extraction for wrapped stdlib loggers. #5

3.5.13 0.3.2 (2013-09-27)

Changes:

- Fix stdlib's name guessing.

3.5.14 0.3.1 (2013-09-26)

Changes:

- Add forgotten `structlog.processors.TimeStamper` to API documentation.

3.5.15 0.3.0 (2013-09-23)

Changes:

- Greatly enhanced and polished the documentation and added a new theme based on Write The Docs, requests, and Flask.
- Add Python Standard Library-specific BoundLogger that has an explicit API instead of intercepting unknown method calls. See `structlog.stdlib.BoundLogger`.
- `structlog.ReturnLogger` now allows arbitrary positional and keyword arguments.
- Add Twisted-specific BoundLogger that has an explicit API instead of intercepting unknown method calls. See `structlog.twisted.BoundLogger`.
- 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 the chapter on performance and the `cache_logger_on_first_use` argument of `structlog.configure` and `structlog.wrap_logger`.

- 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`.

3.5.16 0.2.0 (2013-09-17)

Changes:

- Promote to stable, thus henceforth a strict backward compatibility policy is put into effect.
- Add `key_order` option to `structlog.processors.KeyValueRenderer` for more predictable log entries with any `dict` class.
- `structlog.PrintLogger` now uses proper I/O routines and is thus viable not only for examples but also for production.
- Enhance Twisted support by offering JSONification of non-structlog log entries.
- Allow for custom serialization in `structlog.twisted.JSONRenderer` without abusing `__repr__`.

3.5.17 0.1.0 (2013-09-16)

Initial release.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

P

R

S

T

U

W