structlog Documentation

Release 22.3.0

Author

CONTENTS

Simple. Powerful. Fast. Pick three.

Release **22.3.0** (*What's new?*)

---

structlog is *the* production-ready logging solution for Python:

- **Simple**: At its core, everything is about **functions** that take and return **dictionaries** – all hidden behind **familiar APIs**.

- **Powerful**: Functions and dictionaries aren't just simple, they're also powerful. *structlog* leaves *you* in control.

- **Fast**: *structlog* is not hamstrung by designs of yore. Its flexibility comes not at the price of performance.

Thanks to its flexible design, *you* choose whether you want *structlog* to take care of 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.

The output format is just as flexible and *structlog* comes with support for JSON, *logfmt*, as well as pretty console output out-of-the-box:

```
) python show_off.py
2022-10-15 18:12:27 [debug    ] debugging is hard          a_list=[1, 2, 3]
2022-10-15 18:12:27 [info     ] informative!               some_key=some_value
2022-10-15 18:12:27 [warning  ] uh-uh!
2022-10-15 18:12:27 [error    ] omg                        a_dict={'a': 42, 'b': 'foo'}
2022-10-15 18:12:27 [critical ] wtf                        what=SomeClass(x=1, y='z')
2022-10-15 18:12:27 [error    ] poor me
                             ── Traceback (most recent call last) ─────────
  /Users/hynek/FOSS/structlog/show_off.py:34 in make_call_stack_more_impressive

     31 def make_call_stack_more_impressive():
     32     try:
     33         d = {"x": 42}
   ❱ 34         print(SomeClass(d["y"], "foo"))
     35     except Exception:
     36         log2.exception("poor me")
     37     log.info("all better now!", stack_info=True)

     ── locals ──
     d = {'x': 42}


KeyError: 'y'

2022-10-15 18:12:27 [info     ] all better now!
Stack (most recent call last):
  File "/Users/hynek/FOSS/structlog/show_off.py", line 40, in <module>
    make_call_stack_more_impressive()
  File "/Users/hynek/FOSS/structlog/show_off.py", line 37, in make_call_stack_more_impressive
    log.info("all better now!", stack_info=True)
```

SPONSORS

structlog would not be possible without our amazing sponsors. Especially those generously supporting us at the *The Organization* tier and higher:

---

structlog has been successfully used in production at every scale since **2013**, while embracing cutting-edge technologies like *asyncio*, context variables, or type hints as they emerged. Its paradigms proved influential enough to help design structured logging packages across ecosystems.

If you'd like more information on why structured logging in general – and *structlog* in particular – are good ideas, we've prepared a *summary* just for you.

Otherwise, let's dive right in!

WHY . . .

2.1 . . . 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 log entries. Instead, you log *events* that happen in a *context* of key-value pairs.

Tip: More general advice about production-grade logging can be found in the later chapter on *Logging Best Practices*.

2.2 . . . structlog?

2.2.1 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)
2020-11-18 09:17:09 [info     ] key_value_logging    effort=0 out_of_the_box=True
```

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

That said, *structlog* is not taking anything away from you. You can still use string interpolation using positional arguments:

```
>>> log.info("Hello, %s!", "world")
2022-10-10 07:19:25 [info     ] Hello, world!
```

2.2.2 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)
2020-11-18 09:18:28 [info     ] user.logged_in     another_key=42 happy=True some_key=23␣
→user=hynek
```

You can also bind key-value pairs to *context variables* that look global, but are local to your thread or *asyncio* context (i.e. usually your request).

2.2.3 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:

```python
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?").
- Flexible *timestamping*.

2.2.4 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* of logfmt 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.

2.2.5 Output

structlog is also 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 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 use cases are sending them out via network or saving them to a database.

2.2.6 Highly Testable

structlog is thoroughly tested and we see it as our duty to help you to achieve the same in *your* applications. That's why it ships with a *test helpers* to introspect your application's logging behavior with little-to-no boilerplate.

BASICS

The first chapters teach you all you need to use *structlog* productively. They build gently on each other, so ideally, read them in order.

3.1 Getting Started

3.1.1 Installation

You can install *structlog* from PyPI using *pip*:

```
$ python -m pip install structlog
```

If you want pretty exceptions in development (you know you do!), additionally install either *Rich* or *better-exceptions*. Try both to find out which one you like better – the screenshot in the README and docs homepage is rendered by *Rich*.

On **Windows**, you also have to install *Colorama* if you want colorful output beside exceptions.

3.1.2 Your First Log Entry

A lot of effort went into making *structlog* accessible without reading pages of documentation. As a result, the simplest possible usage looks like this:

```
>>> import structlog
>>> log = structlog.get_logger()
>>> log.info("hello, %s!", "world", key="value!", more_than_strings=[1, 2, 3])
2022-10-07 10:41:29 [info     ] hello, world!   key=value! more_than_strings=[1, 2, 3]
```

Here, *structlog* takes advantage of its default settings:

- Output is sent to **standard out** instead doing nothing.

- It **imitates** standard library `logging`'s **log level names** for familiarity. By default, no level-based filtering is done, but it comes with a **very fast** *filtering machinery*.

- Like in `logging`, positional arguments are **interpolated into the message string using** %. That might look dated, but it's *much* faster than using `str.format` and allows *structlog* to be used as drop-in replacement for `logging`. If you *know* that the log entry is *always* gonna be logged out, just use f-strings which are the fastest.

- All keywords are formatted using `structlog.dev.ConsoleRenderer`. That in turn uses `repr()` to serialize **any value to a string**.

- It's rendered in nice *colors*.

- If you have *Rich* or *better-exceptions* installed, **exceptions** will be rendered in **colors** and with additional **helpful information**.

Please note that even in most complex logging setups the example would still look just like that thanks to *Configuration*. Using the defaults, as above, is equivalent to:

```python
import logging
import structlog

structlog.configure(
    processors=[
        structlog.contextvars.merge_contextvars,
        structlog.processors.add_log_level,
        structlog.processors.StackInfoRenderer(),
        structlog.dev.set_exc_info,
        structlog.processors.TimeStamper(),
        structlog.dev.ConsoleRenderer()
    ],
    wrapper_class=structlog.make_filtering_bound_logger(logging.NOTSET),
    context_class=dict,
    logger_factory=structlog.PrintLoggerFactory(),
    cache_logger_on_first_use=False
)
log = structlog.get_logger()
```

Note:

- *structlog.stdlib.recreate_defaults()* allows you to switch *structlog* to using standard library's `logging` module for output for better interoperability with just one function call.

- *make_filtering_bound_logger()* (re-)uses `logging`'s log levels, but doesn't use `logging` at all. The exposed API is *FilteringBoundLogger*.

- For brevity and to enable doctests, all further examples in *structlog*'s documentation use the more simplistic *KeyValueRenderer()* without timestamps.

Here 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.

3.1.3 Building a Context

Imagine a hypothetical web application that wants to log out all relevant data with just the APIs that we've introduced so far:

```python
def view(request):
    user_agent = request.get("HTTP_USER_AGENT", "UNKNOWN")
    peer_ip = request.client_addr
    if something:
        log.info("something", user_agent=user_agent, peer_ip=peer_ip)
        return "something"
    elif something_else:
        log.info("something_else", user_agent=user_agent, peer_ip=peer_ip)
```

(continues on next page)

```
            return "something_else"
    else:
        log.info("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. It's easy to forget to add a key-value pair in one of the incantations.

At this point, you'll be tempted to write a closure like:

```
def log_closure(event):
    log.info(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? And do you really want to have a logging closure in each of your views?

Let's have a look at a better approach:

```
def view(request):
    log = logger.bind(
        user_agent=request.get("HTTP_USER_AGENT", "UNKNOWN"),
        peer_ip=request.client_addr,
    )

    if foo := request.get("foo"):
        log = log.bind(foo=foo)

    if something:
        log.info("something")
        return "something"
    elif something_else:
        log.info("something_else")
        return "something_else"
    else:
        log.info("else")
        return "else"
```

Suddenly your logger becomes your closure!

To *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 are the `kwargs` of the log call – it is merged together with the *context* to an *event dict* and logged out.

- Each logger with its context is *immutable*. You manipulate the context by creating new loggers using `bind()` and `unbind()`.

The last point is very clean and easy to reason about, but sometimes it's useful to store *some* data globally.

In our example above the peer IP comes to mind. There's no point in extracting it in every view! For that, *structlog* gives you thread-local context storage based on the `contextvars` module:

```
>>> structlog.contextvars.bind_contextvars(peer_ip="1.2.3.4")
>>> structlog.get_logger().info("something")
2022-10-10 10:18:05 [info     ] something     peer_ip=1.2.3.4
```

See *Context Variables* for more information and a more complete example.

3.1.4 Manipulating Log Entries in Flight

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

To facilitate that, *structlog* has the concept of *processor chains*. A processor is 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. Now you have to tell *structlog* about your processor by *configuring* it:

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

3.1.5 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. The type of the return value that is required from the renderer depends on the input that the *logger* that is wrapped by *structlog* needs. While usually it's a string or bytes, there's no rule saying it *has* to be a string!

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().info("hi")
{"event": "hi"}
```

3.1.6 *structlog* and Standard Library's `logging`

While *structlog*'s loggers are very fast and sufficient for the majority of our users, you're not bound to them. Instead, it's been designed from day one to wrap your *existing* loggers and **add** *structure* and *incremental context building* to them.

The most prominent example of such an "existing logger" is certainly the logging module in the standard library. To make this common case as simple as possible, *structlog* comes with *some tools* to help you.

As noted before, the fastest way to transform *structlog* into a `logging`-friendly package is calling `structlog.stdlib.recreate_defaults()`.

3.1.7 asyncio

structlog comes with two approaches to support asynchronous logging.

The default *bound logger* that you get back from `structlog.get_logger()` doesn't have just the familiar log methods like `debug()` or `info()`, but also their async cousins, that simply prefix the name with an a:

```
>>> import asyncio
>>> logger = structlog.get_logger()
>>> async def f():
...     await logger.ainfo("hi!")
...
>>> asyncio.run(f())
2022-10-18 13:23:37 [info     ] hi!
```

You can use the sync and async logging methods interchangeably within the same application.

The standard library integration on the other hand offers an asynchronous wrapper class `structlog.stdlib.AsyncBoundLogger`.

3.1.8 Liked what you saw?

Now you're all set for the rest of the user's guide and can start reading about *bound loggers* – the heart of *structlog*.

3.2 Bound Loggers

The centerpiece of *structlog* that you will interact with most is called a *bound logger*.

It's what you get back from `structlog.get_logger()` and it's called a *bound logger* because you can *bind* key-value pairs to it.

As far as *structlog* is concerned, it consists of three parts:

1. A *context dictionary* that you can *bind* key-value pairs to. This dictionary is *merged* into each log entry that is logged from *this logger specifically*.

   You can inspect a context of a *bound logger* by calling `structlog.get_context()` on it.

2. A list of *processors* that are called on every log entry. Each processor receives the return value of its predecessor passed as an argument.

   This list is usually set using *Configuration*.

---

3. And finally a *logger* that it's wrapping. This wrapped logger is responsible for the *output* of the log entry that has been returned by the last processor. This *can* be standard library's `logging.Logger` like in the image above, but absolutely doesn't have to: By default it's *structlog*'s `PrintLogger`.

   This wrapped logger also is usually set using *Configuration*.

Important: Bound loggers themselves do *not* do any I/O themselves.

All they do is manage the *context* and proxy log calls to a *wrapped logger*.

3.2.1 Context

To manipulate the context dictionary, a *bound logger* can:

- Recreate itself with (optional) *additional* context data: `bind()` and `new()`.
- Recreate itself with *less* context data: `unbind()` and `try_unbind()`.

In any case, the original bound logger or its context are never mutated. They always return a *copy* of the bound logger with a *new* context that reflects your changes.

This part of the API is defined in the `typing.Protocol` called `structlog.typing.BindableLogger`. The protocol is marked `typing.runtime_checkable()` which means that you can check an object for being a *bound logger* using `isinstance(obj, structlog.typing.BindableLogger)`.

3.2.2 Output

Finally, a *bound logger* also **indirectly** exposes the logging methods of the *wrapped logger*. By default, that's a `FilteringBoundLogger` that is wrapping a `PrintLogger`. They both share the set of log methods that's present in the standard library: `debug()`, `info()`, `warning()`, `error()`, and `critical()`.

Whenever you call one of those methods on the *bound logger*, it will:

1. Make a copy of its context – now it becomes the *event dictionary*,
2. Add the keyword arguments of the method call to the event dict.
3. Add a new key `event` with the value of the first positional argument of the method call to the event dict.
4. Run the processors successively on the event dict. Each processor receives the result of its predecessor.
5. Finally, it takes the result of the final processor and calls the method with the same name – that got called on the *bound logger* – on the wrapped logger.

   For flexibility, the final processor can return either a string[1] that is passed directly as a positional parameter, or a tuple `(args, kwargs)` that are passed as `wrapped_logger.log_method(*args, **kwargs)`.

---

[1] `str`, `bytes`, or `bytearray` to be exact.

Step-by-Step Example

Assuming you've left the default configuration and have:

```
import structlog

logger = structlog.get_logger()

log = logger.bind(foo="bar")
```

Now `log` is a *bound logger* of type *FilteringBoundLogger* (but in the default config there's no filtering). `log`'s context is {"foo":   "bar"} and its wrapped logger is a *structlog.PrintLogger*.

Now if you call `log.info("Hello, %s!", "world", number=42)` the following happens:

1. `"world"` gets interpolated into `"Hello, %s!"`, making the event "Hello, world!"[2].

2. The *bound logger*'s context gets copied and the key-value pairs from the `info` call are added to it. It becomes an *event dict* and is {"foo":   "bar", "number":   42} now.

3. The event from step 1 is added too. The *event dict* is {"foo":   "bar", "number":   42, "event":   "Hello, world!"} now.

4. The *event dict* is fed into the *processor chain*. In this case the processors add a timestamp and the log level name to the *event dict*.

   Before it hits the last processor, the *event dict* looks something like {"foo":   "bar", "number":   42, "event":   "Hello, world!", "level":   "info", "timestamp":   "2022-10-13 16:29:27"} now.

   The last processor is *structlog.dev.ConsoleRenderer* and renders the *event dict* into a colorful string[3].

5. Finally, the *wrapped logger*'s (a *PrintLogger*) `info()` method is called with that string.

3.2.3 Filtering by Log Levels

Filtering based on log levels can be done in a processor very easily[4], however that means unnecessary performance overhead through function calls. We care a lot about performance and that's why *structlog*'s default *bound logger* class implements level-filtering as close to the users as possible: in the *bound logger*'s logging methods *before* even creating an *event dict* and starting the processor chain.

structlog.make_filtering_bound_logger() allows you to create a *bound logger* whose log methods with a log level beneath the configured one consist of a plain `return None`.

Here's an example:

```
>>> import structlog
>>> logger = structlog.get_logger()
>>> logger.debug("hi!")
2022-10-15 11:39:03 [debug    ] hi!
>>> import logging
>>> structlog.configure(wrapper_class=structlog.make_filtering_bound_logger(logging.
→INFO))
```

(continues on next page)

---

[2] String interpolation only takes place if you pass positional arguments.
[3] Until this very step, the *event dict* was a dictionary. By replacing the last processor, you decide on the **format** of your logs. For example, if you wanted JSON logs, you just have to replace the last processor with *structlog.processors.JSONRenderer*.
[4] And it's in fact supported for standard library logging with the *structlog.stdlib.filter_by_level()* processor.

```
>>> 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'}
```

3.3 Configuration

The focus of *structlog* has always been to be flexible to a fault. The goal is that a user can use it with *any* logger of their own that is wrapped by *structlog*.

That's the reason why there's an overwhelming amount of knobs to tweak, but – ideally – once you find your configuration, you don't touch it ever again and, more importantly: don't see any of it in your application code.

Let's start at the end and introduce the ultimate convenience function that relies purely on configuration: *structlog. get_logger()*.

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

```
import structlog

logger = structlog.get_logger()
```

To that end, you'll have to call *structlog.configure()* on app initialization. You can call *structlog. configure()* repeatedly and only set one or more settings – the rest will not be affected.

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 using *structlog.is_configured()* and *structlog.get_config()*}:

```
>>> structlog.is_configured()
False
>>> structlog.configure(logger_factory=structlog.stdlib.LoggerFactory)
>>> structlog.is_configured()
True
>>> cfg = structlog.get_config()
>>> cfg["logger_factory"]
<class 'structlog.stdlib.LoggerFactory'>
```

Important: Since you'll call *structlog.get_logger()* in module scope, it runs at import time *before* you had a chance to configure *structlog*. Therefore it returns a **lazy proxy** that returns a correctly configured *bound logger* on its first call to one of the context-managing methods like bind().

Thus, you must never call new() or bind() in module or class scope because , 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 *bound logger* **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*.

3.3.1 What To Configure

You can find the details in the API documentation of *structlog.configure()*, but let's introduce the most important ones at a high level first.

Wrapper Classes

You've met *Bound Loggers* in the last chapter. They're the objects returned by *get_logger()* and allow to bind key-value pairs into their private context. You can configure their type using the `wrapper_class` keyword.

Whenever you bind or unbind data to a *bound logger*, this class is instantiated with the new context and returned.

Logger Factories

We've already talked about wrapped loggers responsible for the output, but until now we haven't explained where they come from until now. Unlike with *bound loggers*, you often need more flexibility when instantiating them. Therefore you don't configure a class; you configure a *factory* using the `logger_factory` keyword.

It's 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.

These will be passed to the logger factories. For example, if you use `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"`.

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 standard library `logging` for output is:

```
>>> 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. See *Standard Library Logging* for more details.

Calling *structlog.get_logger()* without configuration gives you a perfectly useful *structlog.PrintLogger*. We don't believe silent loggers are a sensible default.

Processors

You will meet *Processors* in the next chapter. They are configured using the `processors` keyword that takes an `Iterable` of callables that act as processors.

3.4 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.

3.4.1 Chains

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

logger
> Your wrapped logger object. For example `logging.Logger` or `structlog.typing.FilteringBoundLogger` (default).

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.

Note: *structlog* only looks at the return value of the **last** processor. That means that as long as you control the next processor in the chain (i.e. the processor that will get your return value passed as an argument), you can return whatever you want.

Returning a modified event dictionary from your processors is just a convention to make processors composable.

Examples

If you set up your logger like:

```
structlog.configure(processors=[f1, f2, f3])
log = structlog.get_logger().bind(x=42)
```

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

```
wrapped_logger.info(
    f3(wrapped_logger, "info",
        f2(wrapped_logger, "info",
            f1(wrapped_logger, "info", {"event": "some_event", "x": 42, "y": 23})
        )
    )
)
```

In this case, `f3` has to make sure it returns something `wrapped_logger.info` can handle (see *Adapting and Rendering*). For the example with `PrintLogger` above, this means `f3` must return a string.

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:

```python
import calendar
import time


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

Important: You're explicitly allowed to modify the `event_dict` parameter, because a copy has been created before calling the first processor.

Please note that *structlog* comes with such a processor built in: *TimeStamper*.

3.4.2 Filtering

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

Therefore, the following processor drops every entry:

```python
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)?

```python
class ConditionalDropper:
    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

        return event_dict
```

Since it's so common to filter by the log level, *structlog* comes with `structlog.make_filtering_bound_logger()` that filters log entries before they even enter the processor chain. It does **not** use the standard library, but it does use its

names and order of log levels.

3.4.3 Adapting and Rendering

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

It can return one of three types:

- An Unicode string (`str`), a bytes string (`bytes`), or a `bytearray` 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`.

Changed in version 20.2.0: Allow final processor to return a `bytes`.

Changed in version 21.2.0: Allow final processor to return a `bytearray`.

Examples

The probably most useful formatter for string based loggers is `structlog.processors.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.

For a list of shipped processors, check out the *API documentation*.

3.4.4 Third-Party Packages

structlog was specifically designed to be as composable and reusable as possible, so whatever you're missing: chances are, you can solve it with a processor! Since processors are self-contained callables, it's easy to write your own and to share them in separate packages.

We collect those packages in our GitHub Wiki and encourage you to add your package too!

3.5 Context Variables

The `contextvars` module in the Python standard library allows having a global *structlog* context that is local to the current execution context. The execution context can be thread-local if using threads, or using primitives based on `asyncio`, or *greenlet* respectively.

For example, you may want to bind certain values like a request ID or the peer's IP address at the beginning of a web request and have them logged out along with the local contexts you build within our views.

For that *structlog* provides the `structlog.contextvars` module with a set of functions to bind variables to a context-local context. This context is safe to be used both in threaded as well as asynchronous code.

The general flow is:

- Use *structlog.configure()* with *structlog.contextvars.merge_contextvars()* as your first processor (part of default configuration).

- Call *structlog.contextvars.clear_contextvars()* at the beginning of your request handler (or whenever you want to reset the context-local context).

- Call *structlog.contextvars.bind_contextvars()* and *structlog.contextvars.unbind_contextvars()* instead of your bound logger's `bind()` and `unbind()` when you want to bind and unbind key-value pairs to the context-local context. You can also use the *structlog.contextvars.bound_contextvars()* context manager / decorator.

- Use *structlog* as normal. Loggers act as they always do, but the *structlog.contextvars.merge_contextvars()* processor ensures that any context-local binds get included in all of your log messages.

- If you want to access the context-local storage, you use *structlog.contextvars.get_contextvars()* and *structlog.contextvars.get_merged_contextvars()*.

We're sorry the word *context* means three different things in this itemization depending on … context.

```
>>> from structlog.contextvars import (
...     bind_contextvars,
...     bound_contextvars,
...     clear_contextvars,
...     merge_contextvars,
...     unbind_contextvars,
... )
>>> from structlog import configure
>>> configure(
...     processors=[
...         merge_contextvars,
...         structlog.processors.KeyValueRenderer(key_order=["event", "a"]),
...     ]
... )
>>> log = structlog.get_logger()
>>> # At the top of your request handler (or, ideally, some general
>>> # middleware), clear the contextvars-local context and bind some common
>>> # values:
>>> clear_contextvars()
>>> bind_contextvars(a=1, b=2)
{'a': <Token var=<ContextVar name='structlog_a' default=Ellipsis at ...> at ...>, 'b':
→<Token var=<ContextVar name='structlog_b' default=Ellipsis at ...> at ...>}
>>> # Then use loggers as per normal
>>> # (perhaps by using structlog.get_logger() to create them).
>>> log.info("hello")
event='hello' a=1 b=2
>>> # Use unbind_contextvars to remove a variable from the context.
>>> unbind_contextvars("b")
>>> log.info("world")
event='world' a=1
>>> # You can also bind key-value pairs temporarily.
>>> with bound_contextvars(b=2):
...     log.info("hi")
event='hi' a=1 b=2
>>> # Now it's gone again.
```

```
>>> log.info("hi")
event='hi' a=1
>>> # And when we clear the contextvars state again, it goes away.
>>> # a=None is printed due to the key_order argument passed to
>>> # KeyValueRenderer, but it is NOT present anymore.
>>> clear_contextvars()
>>> log.info("hi there")
event='hi there' a=None
```

3.5.1 Support for `contextvars.Token`

If e.g. your request handler calls a helper function that needs to temporarily override some contextvars before restoring them back to their original values, you can use the Tokens returned by `bind_contextvars()` along with `reset_contextvars()` to accomplish this (much like how `contextvars.ContextVar.reset()` works):

```python
def foo():
    bind_contextvars(a=1)
    _helper()
    log.info("a is restored!")  # a=1

def _helper():
    tokens = bind_contextvars(a=2)
    log.info("a is overridden")  # a=2
    reset_contextvars(**tokens)
```

3.5.2 Example: Flask and Thread-Local Data

Let's assume you want to bind a unique request ID, the URL path, and the peer's IP to every log entry by storing it in thread-local storage that is managed by context variables:

```python
import logging
import sys
import uuid

import flask

from .some_module import some_function

import structlog

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

@app.route("/login", methods=["POST", "GET"])
def some_route():
    # You would put this into some kind of middleware or processor so it's set
    # automatically for all requests in all views.
    structlog.contextvars.clear_contextvars()
    structlog.contextvars.bind_contextvars(
        view=flask.request.path,
```

```python
        request_id=str(uuid.uuid4()),
        peer=flask.request.access_route[0],
    )
    # End of belongs-to-middleware.

    log = logger.bind()
    # do something
    # ...
    log.info("user logged in", 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.contextvars.merge_contextvars,  # <--!!!
            structlog.processors.KeyValueRenderer(
                key_order=["event", "view", "peer"]
            ),
        ],
        logger_factory=structlog.stdlib.LoggerFactory(),
    )
    app.run()
```

`some_module.py`:

```python
from structlog import get_logger

logger = get_logger()

def some_function():
    # ...
    logger.error("user did something", something="shot_in_foot")
    # ...
```

This would result among other the following lines to be printed:

```
event='user logged in' view='/login' peer='127.0.0.1' user='test-user' request_id=
↪'e08ddf0d-23a5-47ce-b20e-73ab8877d736'
event='user did something' view='/login' peer='127.0.0.1' something='shot_in_foot'␣
↪request_id='e08ddf0d-23a5-47ce-b20e-73ab8877d736'
```

As you can see, `view`, `peer`, and `request_id` are present in **both** log entries.

DEVELOPMENT AFFORDANCES

structlog's focus is production systems, but it comes with **pretty console logging** and handy in-development helpers both for your **comfort** and your code's **quality**.

4.1 Console Output

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[1] console output.

If either of the *Rich* or *better-exceptions* packages is installed, it will also pretty-print exceptions with helpful contextual data. *Rich* takes precedence over *better-exceptions*, but you can configure it by passing `structlog.dev.plain_traceback()` or `structlog.dev.better_traceback()` for the `exception_formatter` parameter of `ConsoleRenderer`.

The following output is rendered using *Rich*:

You can find the code for the output above in the repo.

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 `exc_info` as produced by *structlog*'s processors and render them in special ways.

> **Warning:** For pretty exceptions to work, `format_exc_info()` must be **absent** from the processors chain.

structlog's default configuration already uses `ConsoleRenderer`, therefore if you want nice colorful output on the console, you don't have to do anything except installing *Rich* or *better-exceptions* (and *Colorama* on Windows). If you want to use it along with standard library logging, we suggest the following configuration:

```python
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.dev.ConsoleRenderer()  # <===
    ],
```

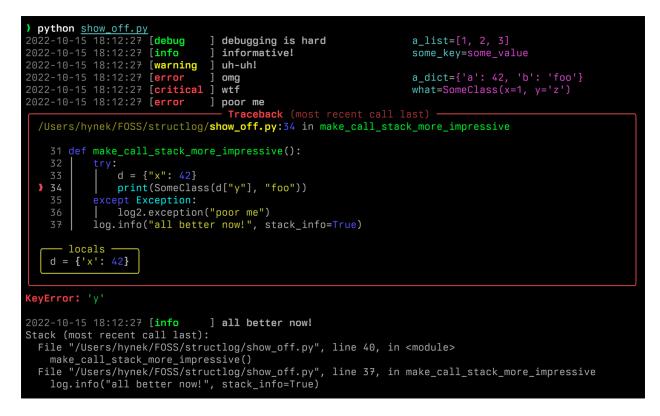(continues on next page)

---

[1] Requires the *Colorama* package on Windows.

Fig. 1: Colorful console output by ConsoleRenderer.

```
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    wrapper_class=structlog.stdlib.BoundLogger,
    cache_logger_on_first_use=True,
)
```

4.1.1 Disabling Exception Pretty-Printing

If you prefer the default terse Exception rendering, but still want *Rich* installed, you can disable the pretty-printing by instantiating *structlog.dev.ConsoleRenderer()* yourself and passing exception_formatter=structlog.dev.plain_traceback.

4.2 Testing

structlog comes with tools for testing the logging behavior of your application.

If you need functionality similar to unittest.TestCase.assertLogs(), or you want to capture all logs for some other reason, you can use the *structlog.testing.capture_logs()* context manager:

```
>>> from structlog import get_logger
>>> from structlog.testing import capture_logs
>>> with capture_logs() as cap_logs:
...     get_logger().bind(x="y").info("hello")
>>> cap_logs
[{'x': 'y', 'event': 'hello', 'log_level': 'info'}]
```

Note that inside the context manager all configured processors are disabled.

Note: capture_logs() relies on changing the configuration. If you have *cache_logger_on_first_use* enabled for *performance*, any cached loggers will not be affected, so it's recommended you do not enable it during tests.

You can build your own helpers using *structlog.testing.LogCapture*. For example a *pytest* fixture to capture log output could look like this:

```
@pytest.fixture(name="log_output")
def fixture_log_output():
    return LogCapture()

@pytest.fixture(autouse=True)
def fixture_configure_structlog(log_output):
    structlog.configure(
        processors=[log_output]
    )

def test_my_stuff(log_output):
    do_something()
    assert log_output.entries == [...]
```

You can also use *structlog.testing.CapturingLogger* (directly, or via *CapturingLoggerFactory* that always returns the same logger) that is more low-level and great for unit tests:

```
>>> import structlog
>>> cf = structlog.testing.CapturingLoggerFactory()
>>> structlog.configure(logger_factory=cf, processors=[structlog.processors.
→JSONRenderer()])
>>> log = get_logger()
>>> log.info("test!")
>>> cf.logger.calls
[CapturedCall(method_name='info', args=('{"event": "test!"}',), kwargs={})]
```

Additionally *structlog* also ships with a logger that just returns whatever it gets passed into it: *structlog.testing.ReturnLogger*.

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

4.3 Type Hints

Static type hints – together with a type checker like *Mypy* – are an excellent way to make your code more robust, self-documenting, and maintainable in the long run. And as of 20.2.0, *structlog* comes with type hints for all of its APIs.

Since *structlog* is highly configurable and tries to give a clean façade to its users, adding types without breaking compatibility – while remaining useful! – was a formidable task.

The main problem is that `structlog.get_logger()` returns whatever you've configured the *bound logger* to be. The only commonality are the binding methods like `bind()` and we've extracted them into the `structlog.typing.BindableLogger` `Protocol`. But using that as a return type is worse than useless, because you'd have to use `typing.cast()` on every logger returned by `structlog.get_logger()`, if you wanted to actually call any logging methods.

The second problem is that said `bind()` and its cousins are inherited from a common base class (a big mistake in hindsight) and can't know what concrete class subclasses them and therefore what type they are returning.

The chosen solution is adding `structlog.stdlib.get_logger()` that just calls `structlog.get_logger()` but has the correct type hints and adding `structlog.stdlib.BoundLogger.bind` et al that also only delegate to the base class.

`structlog.get_logger()` is typed as returning `typing.Any` so you can use your own type annotation and stick to the old APIs, if that's what you prefer:

```python
import structlog

logger: structlog.stdlib.BoundLogger = structlog.get_logger()
logger.info("hi")  # <- ok
logger.msg("hi")   # <- Mypy: 'error: "BoundLogger" has no attribute "msg"'
```

FIVE

INTEGRATION WITH EXISTING SYSTEMS

structlog is both zero-config as well as highly configurable. You can use it on its own or integrate with existing systems. Dedicated support for the standard library and Twisted is shipped out-of-the-box.

5.1 Frameworks

To have consistent log output, it makes sense to configure *structlog before* any logging is done. The best place to perform your configuration varies with applications and frameworks. If you use standard library's logging, it makes sense to configure them next to each other.

5.1.1 OpenTelemetry

The Python *OpenTelemetry* SDK offers an easy API to get the current span, so you can enrich your logs with a straight-forward processor:

```python
from opentelemetry import trace

def add_open_telemetry_spans(_, __, event_dict):
    span = trace.get_current_span()
    if not span.is_recording():
        event_dict["span"] = None
        return event_dict

    ctx = span.get_span_context()
    parent = getattr(span, "parent", None)

    event_dict["span"] = {
        "span_id": hex(ctx.span_id),
        "trace_id": hex(ctx.trace_id),
        "parent_span_id": None if not parent else hex(parent.span_id),
    }

    return event_dict
```

5.1.2 Django

django-structlog is a popular and well-maintained package that does all the heavy lifting.

5.1.3 Flask

See Flask's Logging docs.

Generally speaking: configure *structlog before* instantiating `flask.Flask`.

Here's a signal handler that binds various request details into context variables:

```python
def bind_request_details(sender: Flask, **extras: dict[str, Any]) -> None:
    structlog.contextvars.clear_contextvars()
    structlog.contextvars.bind_contextvars(
        request_id=request.headers.get("X-Unique-ID", "NONE"),
        peer=peer,
    )

    if current_user.is_authenticated:
        structlog.contextvars.bind_contextvars(
            user_id=current_user.get_id(),
        )
```

You add it to an existing `app` like this:

```python
from flask import request_started

request_started.connect(bind_request_details, app)
```

5.1.4 Pyramid

Configure it in the application constructor.

Here's an example for a *Pyramid Tween* that stores various request-specific data into context variables:

```python
@dataclass
class StructLogTween:
    handler: Callable[[Request], Response]
    registry: Registry

    def __call__(self, request: Request) -> Response:
        structlog.contextvars.clear_contextvars()
        structlog.contextvars.bind_contextvars(
            peer=request.client_addr,
            request_id=request.headers.get("X-Unique-ID", "NONE"),
            user_agent=request.environ.get("HTTP_USER_AGENT", "UNKNOWN"),
            user=request.authenticated_userid,
        )

        return self.handler(request)
```

5.1.5 Celery

Celery's multi-process architecture leads unavoidably to race conditions that show up as interleaved logs. It ships standard library-based helpers in the form of `celery.utils.log.get_task_logger()` that you should use inside of tasks to prevent that problem.

The most straight-forward way to integrate that with *structlog* is using *Standard Library Logging* and wrapping that logger using `structlog.wrap_logger()`:

```python
from celery.utils.log import get_task_logger

logger = structlog.wrap_logger(get_task_logger(__name__))
```

If you want to automatically bind task metadata to your *Context Variables*, you can use *Celery*'s signals:

```python
from celery import signals

@signals.task_prerun.connect
def on_task_prerun(sender, task_id, task, args, kwargs, **_):
    structlog.contextvars.bind_contextvars(task_id=task_id, task_name=task.name)
```

See this issue for more details.

5.1.6 Twisted

The plugin definition is the best place. If your app is not a plugin, put it into your tac file.

5.2 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!

Important: The quickest way to get started with *structlog* and `logging` is `structlog.stdlib.recreate_defaults()`. It will recreate the default configuration on top of `logging` and optionally configure `logging` for you.

5.2.1 Just Enough `logging`

If you want to use *structlog* with `logging`, you should have at least a 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 to *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.

5.2.2 Concrete Bound Logger

structlog ships a stdlib-specific bound logger that mirrors the log methods of standard library's `logging.Logger` with correct type hints.

If you want to take advantage of said type hints, you have to either annotate the logger coming from *structlog. get_logger()*, or use *structlog.stdlib.get_logger()* that has the appropriate type hints. Please note though, that it will neither configure nor verify your configuration. It will call `structlog.get_logger()` just like if you would've called it – the only difference are the type hints.

See also *Type Hints*.

asyncio

For `asyncio` applications, you may not want your whole application to block while your processor chain is formatting your log entries. For that use case *structlog* comes with *structlog.stdlib.AsyncBoundLogger* that will do all processing in a thread pool executor.

This means an increased computational cost per log entry but your application will never block because of logging.

To use it, *configure structlog* to use `AsyncBoundLogger` as `wrapper_class`.

5.2.3 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 from happening in the first place.

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

ExtraAdder():
> Add extra attributes of `logging.LogRecord` objects to the event dictionary.

This processor can be used for adding data passed in the `extra` parameter of the `logging` module's log methods to the event dictionary.

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

add_log_level_number():
  Adds the log level number to the event dictionary under the key `level_number`. Log level numbers map to the log level names. The Python stdlib uses them for filtering logic. This adds the same numbers so users can leverage similar filtering. Compare:

```
level in ("warning", "error", "critical")
level_number >= 30
```

  The mapping of names to numbers is in `structlog.stdlib._NAME_TO_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).

5.2.4 Suggested Configurations

Note: We do appreciate that fully integrating *structlog* with standard library's `logging` is fiddly when done for the first time.

This is the price of flexibility and unfortunately – given the different needs of our users – we can't make it any simpler without compromising someone's use-cases. However, once it is set up, you can rely on not having to ever touch it again.

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

Don't Integrate

The most straight-forward option is to configure standard library `logging` close enough to what *structlog* is logging and leaving it at that.

Since these are usually log entries from third parties that don't take advantage of *structlog*'s features, this is surprisingly often a perfectly adequate approach.

For instance, if you log JSON in production, configure `logging` to use *python-json-logger* to make it print JSON too, and then tweak the configuration to match their outputs.

Rendering Within *structlog*

This is the simplest approach where *structlog* does all the heavy lifting and passes a fully-formatted string to `logging`. Chances are, this is all you need.

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

```python
import structlog

structlog.configure(
    processors=[
        # If log level is too low, abort pipeline and throw away log entry.
        structlog.stdlib.filter_by_level,
        # Add the name of the logger to event dict.
        structlog.stdlib.add_logger_name,
        # Add log level to event dict.
        structlog.stdlib.add_log_level,
        # Perform %-style formatting.
        structlog.stdlib.PositionalArgumentsFormatter(),
        # Add a timestamp in ISO 8601 format.
        structlog.processors.TimeStamper(fmt="iso"),
        # If the "stack_info" key in the event dict is true, remove it and
        # render the current stack trace in the "stack" key.
        structlog.processors.StackInfoRenderer(),
        # If the "exc_info" key in the event dict is either true or a
        # sys.exc_info() tuple, remove "exc_info" and render the exception
        # with traceback into the "exception" key.
        structlog.processors.format_exc_info,
        # If some value is in bytes, decode it to a unicode str.
        structlog.processors.UnicodeDecoder(),
        # Add callsite parameters.
        structlog.processors.CallsiteParameterAdder(
            {
                structlog.processors.CallsiteParameter.FILENAME,
                structlog.processors.CallsiteParameter.FUNC_NAME,
                structlog.processors.CallsiteParameter.LINENO,
            }
        ),
        # Render the final event dict as JSON.
        structlog.processors.JSONRenderer()
    ],
    # `wrapper_class` is the bound logger that you get back from
    # get_logger(). This one imitates the API of `logging.Logger`.
    wrapper_class=structlog.stdlib.BoundLogger,
    # `logger_factory` is used to create wrapped loggers that are used for
    # OUTPUT. This one returns a `logging.Logger`. The final value (a JSON
    # string) from the final processor (`JSONRenderer`) will be passed to
    # the method of the same name as that you've called on the bound logger.
    logger_factory=structlog.stdlib.LoggerFactory(),
    # Effectively freeze configuration after creating the first bound
    # logger.
    cache_logger_on_first_use=True,
)
```

To make your program behave like a proper *12 Factor App* that outputs only JSON to `stdout`, configure the `logging`

module like this:

```python
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:

```python
>>> 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
```

Rendering Using `logging`-based Formatters

You can choose to use *structlog* only for building the event dictionary and leave all formatting – additionally to the output – to the standard library.

```python
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(),
        # Transform event dict into `logging.Logger` method arguments.
        # "event" becomes "msg" and the rest is passed as a dict in
        # "extra". IMPORTANT: This means that the standard library MUST
        # render "extra" for the context to appear in log entries! See
        # warning below.
        structlog.stdlib.render_to_log_kwargs,
    ],
    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 application / *structlog*) to be formatted as JSON, you can use the *python-json-logger* library:

```python
import logging
import sys
```

```python
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:

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

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

> **Warning:** With this approach, it's the standard library `logging` formatter's duty to do something useful with the event dict. In the above example that's `jsonlogger.JsonFormatter`.
>
> Keep this in mind if you only get the event name without any context, and exceptions are ostensibly swallowed.

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

Finally, the most ambitious approach. Here, you use *structlog*'s *ProcessorFormatter* as a `logging.Formatter` for both `logging` as well as *structlog* log entries.

Consequently, the output is the duty of the standard library too.

ProcessorFormatter has two parts to its API:

1. On the *structlog* side, the *processor chain* must be configured to end with `structlog.stdlib.ProcessorFormatter.wrap_for_formatter()` as the renderer. It converts the processed event dictionary into something that `ProcessorFormatter` understands.

2. On the `logging` side, you must configure `ProcessorFormatter` as your formatter of choice. `logging` then calls `ProcessorFormatter`'s `format()` method.

   For that, `ProcessorFormatter` wraps a processor chain that is responsible for rendering your log entries to strings.

Thus, the simplest possible configuration looks like the following:

```python
import logging
import structlog

structlog.configure(
    processors=[
        # Prepare event dict for `ProcessorFormatter`.
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    logger_factory=structlog.stdlib.LoggerFactory(),
)
```

```
formatter = structlog.stdlib.ProcessorFormatter(
    processors=[structlog.dev.ConsoleRenderer()],
)


handler = logging.StreamHandler()
# Use OUR `ProcessorFormatter` to format all `logging` entries.
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      _from_structlog=False _record=<LogRecord:...>
>>> structlog.get_logger("structlog").info("amazing", events="oh yes")
amazing  _from_structlog=True _record=<LogRecord:...> 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.

`_from_structlog` and `_record` allow your processors to determine whether the log entry is coming from *structlog*, and to extract information from `logging.LogRecord`s and add them to the event dictionary. However, you probably don't want to have them in your log files, thus we've added the `ProcessorFormatter.remove_processors_meta` processor to do so conveniently.

For example, to add timestamps, log levels, and traceback handling to your logs without `_from_structlog` and `_record` noise 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(
    # These run ONLY on `logging` entries that do NOT originate within
    # structlog.
    foreign_pre_chain=shared_processors,
    # These run on ALL entries after the pre_chain is done.
```

```
    processors=[
        # Remove _record & _from_structlog.
        structlog.stdlib.ProcessorFormatter.remove_processors_meta,
        structlog.dev.ConsoleRenderer(),
    ],
)
```

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

```
>>> logging.getLogger("stdlog").info("woo")
2021-11-15 11:41:47 [info     ] woo
>>> structlog.get_logger("structlog").info("amazing", events="oh yes")
2021-11-15 11:41:47 [info     ] amazing      events=oh yes
```

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

```python
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,
    # Add extra attributes of LogRecord objects to the event dictionary
    # so that values passed in the extra parameter of log methods pass
    # through to log output.
    structlog.stdlib.ExtraAdder(),
    timestamper,
]

def extract_from_record(_, __, event_dict):
    """
    Extract thread and process names and add them to the event dict.
    """
    record = event_dict["_record"]
    event_dict["thread_name"] = record.threadName
    event_dict["process_name"] = record.processName

    return event_dict

logging.config.dictConfig({
        "version": 1,
        "disable_existing_loggers": False,
        "formatters": {
            "plain": {
                "()": structlog.stdlib.ProcessorFormatter,
                "processors": [
                    structlog.stdlib.ProcessorFormatter.remove_processors_meta,
                    structlog.dev.ConsoleRenderer(colors=False),
                ],
```

```
                "foreign_pre_chain": pre_chain,
            },
            "colored": {
                "()": structlog.stdlib.ProcessorFormatter,
                "processors": [
                    extract_from_record,
                    structlog.stdlib.ProcessorFormatter.remove_processors_meta,
                    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,
    ],
    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()*.

Additionally, for both `logging` and *structlog* – but only for the colorful logger – we also extract some data from `logging.LogRecord`:

```
>>> logging.getLogger().warning("bar")
2021-11-15 13:26:52 [warning  ] bar      process_name=MainProcess thread_name=MainThread

>>> structlog.get_logger("structlog").warning("foo", x=42)
2021-11-15 13:26:52 [warning  ] foo      process_name=MainProcess thread_name=MainThread␣
↪x=42

>>> pathlib.Path("test.log").read_text()
2021-11-15 13:26:52 [warning  ] bar
2021-11-15 13:26:52 [warning  ] foo      x=42
```

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

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

5.3 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`.

5.3.1 Concrete Bound Logger

To make *structlog*'s behavior less magical, 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`.

5.3.2 Processors

structlog comes with two Twisted-specific processors:

structlog.twisted.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-happened")
```

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:

structlog.twisted.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.

5.3.3 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: `structlog.twisted.JSONLogObserverWrapper`.

What it does is determining whether a log entry has been formatted by `structlog.twisted.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 `structlog.twisted.PlainFileLogObserver` that writes only the plain message to a file and `structlog.twisted.plainJSONStdOutLogger` that composes it with the aforementioned `structlog.twisted.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 ...>", ...}
...
```

5.3.4 Suggested Configuration

```python
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*.

STRUCTLOG IN PRACTICE

The following chapters deal with considerations of using *structlog* in the real world.

6.1 Recipes

Thanks to the fact that *structlog* is entirely based on dictionaries and callables, the sky is the limit with what you an achieve. In the beginning that can be daunting, so here are a few examples of tasks that have come up repeatedly.

Please note that recipes related to integration with frameworks have an *own chapter*.

6.1.1 Renaming the event Key

The name of the event is hard-coded in *structlog* to event. But that doesn't mean it has to be called that in your logs.

With the *structlog.processors.EventRenamer* processor you can for instance rename the log message to msg and use event for something custom, that you bind to _event in your code:

```
>>> from structlog.processors import EventRenamer
>>> event_dict = {"event": "something happened", "_event": "our event!"}
>>> EventRenamer("msg", "_event")(None, None, event_dict)
{'msg': 'something happened', 'event': 'our event!'}
```

6.1.2 Fine-Grained Log-Level Filtering

structlog's native log levels as provided by *structlog.make_filtering_bound_logger()* only know **one** log level – the one that is passed to make_filtering_bound_logger(). Sometimes, that can be a bit too coarse, though.

You can achieve finer control by adding the *CallsiteParameterAdder* processor and writing a simple processor that acts on the call site data added.

Let's assume you have the following code:

```python
logger = structlog.get_logger()

def f():
    logger.info("f called")

def g():
    logger.info("g called")
```

```
f()
g()
```

And you don't want to see log entries from function f. You add `CallsiteParameterAdder` to the processor chain and then look at the func_name field in the *event dict*:

```python
def filter_f(_, __, event_dict):
    if event_dict.get("func_name") == "f":
        raise structlog.DropEvent

    return event_dict

structlog.configure(
    processors=[
        structlog.processors.CallsiteParameterAdder(
            [structlog.processors.CallsiteParameter.FUNC_NAME]
        ),
        filter_f,  # <-- your processor!
        structlog.processors.KeyValueRenderer(),
    ]
)
```

Running this gives you:

```
event='g called' func_name='g'
```

`CallsiteParameterAdder` is *very* powerful in what info it can add, so your possibilities are limitless. Pick the data you're interested in from the `structlog.processors.CallsiteParameter` Enum.

6.1.3 Custom Wrappers

The type of the *bound loggers* that are returned by `structlog.get_logger()` is called the *wrapper class*, because it wraps the original logger that takes care of the output. This wrapper class is *configurable*.

Originally, *structlog* used a generic wrapper class `structlog.BoundLogger` by default. That class still ships with *structlog* and can wrap *any* logger class by intercepting unknown method names and proxying them to the wrapped logger.

Nowadays, the default is a `structlog.typing.FilteringBoundLogger` that imitates standard library's log levels with the possibility of efficiently filtering at a certain level (inactive log methods are a plain `return None` each).

If you're integrating with `logging` or Twisted, you may was to use one of their specific *bound loggers* (`structlog.stdlib.BoundLogger` and `structlog.twisted.BoundLogger`, respectively).

—

On top of that all, you can also write your own wrapper classes. To make it easy for you, *structlog* 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 easiest to demonstrate with an example:

```
>>> from structlog import BoundLoggerBase, PrintLogger, wrap_logger
>>> class SemanticLogger(BoundLoggerBase):
...     def info(self, event, **kw):
...         if not "status" in kw:
...             return self._proxy_to_logger("info", event, status="ok", **kw)
...         else:
...             return self._proxy_to_logger("info", event, **kw)
...
...     def user_error(self, event, **kw):
...         self.info(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 `structlog.DropEvent`s and calls a named function on `_logger`.

- You can run the chain by hand through using `structlog.BoundLoggerBase._process_event()`.

These two methods and one attribute are all you need to write own *bound loggers*.

6.1.4 Passing Context to Worker Threads

Thread-local context data based on *context variables* is – as the name says – local to the thread that binds it. When using threads to process work in parallel, you have to pass the thread-local context **into** the worker threads. One way is to retrieve the context vars and pass them along to the worker threads. Then, Inside of the worker, re-bind them using `bind_contextvars`.

The following example uses *pathos* to create a `ThreadPool`. The context variables are retrieved and passed as the first argument to the partial function. The pool invokes the partial function, once for each element of `workers`. Inside of `do_some_work`, the context vars are bound and a message about the great work being performed is logged – including the `request_id` key / value pair.

```python
from functools import partial

import structlog

from structlog.contextvars import bind_contextvars
from pathos.threading import ThreadPool

logger = structlog.get_logger(__name__)


def do_some_work(ctx, this_worker):
    bind_contextvars(**ctx)
```

(continues on next page)

```python
    logger.info("WorkerDidSomeWork", worker=this_worker)


def structlog_with_threadpool(f):
    ctx = structlog.contextvars.get_contextvars()
    func = partial(f, ctx)
    workers = ["1", "2", "3"]

    with ThreadPool() as pool:
        return list(pool.map(func, workers))


def manager(request_id: str):
    bind_contextvars(request_id=request_id)
    logger.info("StartingWorkers")
    structlog_with_threadpool(do_some_work)
```

See the issue 425 for a more complete example.

6.2 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[1].

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 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.

6.2.1 Canonical Log Lines

Generally speaking, having as few log entries per request as possible is a good thing. The less noise, the more insights.

structlog's ability to *bind data to loggers incrementally* – plus *loggers that are local to the current execution context* – can help you to minimize the output to a *single log entry*.

At Stripe, this concept is called Canonical Log Lines.

---

[1] 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.

6.2.2 Pretty Printing vs. Structured Output

Colorful and pretty printed log messages are nice during development when you locally run your code.

However, in production you should emit structured output (like JSON) which is a lot easier to parse by log aggregators. Since you already log in a structured way, writing JSON output with *structlog* comes naturally. You can even generate structured exception tracebacks. This makes analyzing errors easier, since log aggregators can render JSON much better than multiline strings with a lot escaped quotation marks.

Here is a simple example of how you can have pretty logs during development and JSON output when your app is running in a production context:

```python
>>> import sys
>>> import structlog
>>>
>>> shared_processors = [
...     # Processors that have nothing to do with output,
...     # e.g., add timestamps or log level names.
... ]
>>> if sys.stderr.isatty():
...     # Pretty printing when we run in a terminal session.
...     # Automatically prints pretty tracebacks when "rich" is installed
...     processors = shared_processors + [
...         structlog.dev.ConsoleRenderer(),
...     ]
... else:
...     # Print JSON when we run, e.g., in a Docker container.
...     # Also print structured tracebacks.
...     processors = shared_processors + [
...         structlog.processors.dict_tracebacks,
...         structlog.processors.JSONRenderer(),
...     ]
>>> structlog.configure(processors)
```

6.2.3 Centralized Logging

Nowadays you usually don't want your log files in compressed archives distributed over dozens – if not thousands – of 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 *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* either that your log entries are prepended with a timestamp from `TimeStamper` or the name of your timestamp field.

Graylog

Graylog goes one step further. It not only supports everything those above do (and then some); you can also directly log 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*.

6.3 Performance

Here are a few hints how to get the best performance out of *structlog* in production:

- Use *structlog*'s native *BoundLogger* (created using `structlog.make_filtering_bound_logger()`) if you want to use level-based filtering. `return None` is hard to beat.

- Avoid (frequently) calling log methods on loggers you get back from `structlog.get_logger()` or `structlog.wrap_logger()`. Since those functions are usually called in module scope and thus before you are able to configure them, they return a proxy object 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)
  ```

  Since global scope lookups are expensive in Python, it's generally a good idea to copy frequently-used symbols into local scope.

- 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 two drawbacks:

  1. Later calls of `configure()` don't have any effect on already cached loggers – that shouldn't matter outside of *testing* though.

  2. The resulting bound logger is not pickleable. Therefore, you can't set this option if you e.g. plan on passing loggers around using `multiprocessing`.

- Avoid sending your log entries through the standard library if you can: its dynamic nature and flexibility make it a major bottleneck. Instead use `structlog.WriteLoggerFactory` or – if your serializer returns bytes (e.g. *orjson*) – `structlog.BytesLoggerFactory`.

  You can still configure `logging` for packages that you don't control, but avoid it for your *own* log entries.

- Use a faster JSON serializer than the standard library. Possible alternatives are among others are *orjson* or *RapidJSON*.

- Be conscious about whether and how you use *structlog*'s *asyncio* support. While it's true that moving log processing into separate threads prevents your application from hanging, it also comes with a performance cost.

  Decide judiciously whether or not you're willing to pay that price. If your processor chain has a good and predictable performance without external dependencies (as it should), it might not be worth it.

6.3.1 Example

Here's an example for a production-ready non-*asyncio structlog* configuration that's as fast as it gets:

```python
import logging
import structlog

structlog.configure(
    cache_logger_on_first_use=True,
    wrapper_class=structlog.make_filtering_bound_logger(logging.INFO),
    processors=[
        structlog.contextvars.merge_contextvars,
        structlog.processors.add_log_level,
        structlog.processors.format_exc_info,
        structlog.processors.TimeStamper(fmt="iso", utc=True),
        structlog.processors.JSONRenderer(serializer=orjson.dumps),
    ],
    logger_factory=structlog.BytesLoggerFactory(),
)
```

It has the following properties:

- Caches all loggers on first use.

- Filters all log entries below the `info` log level **very** efficiently. The `debug` method literally consists of `return None`.

- Supports *Context Variables* (thread-local contexts).

- Adds the log level name.

- Renders exceptions.

- Adds an ISO 8601 timestamp under the `timestamp` key in the UTC timezone.

- Renders the log entries as JSON using *orjson* which is faster than plain logging in `logging`.

- Uses `structlog.BytesLoggerFactory` because *orjson* returns bytes. That saves encoding ping-pong.

Therefore a log entry might look like this:

```json
{"event":"hello","timestamp":"2020-11-17T09:54:11.900066Z"}
```

If you need standard library support for external projects, you can either just use a JSON formatter like *python-json-logger*, or pipe them through *structlog* as documented in *Standard Library Logging*.

DEPRECATED FEATURES

7.1 Legacy Thread-local Context

> **Attention:** The `structlog.threadlocal` module is deprecated as of *structlog* 22.1.0 in favor of *Context Variables*.
>
> The standard library `contextvars` module provides a more feature-rich superset of the thread-local APIs and works with thread-local data, async code, and greenlets.
>
> Therefore, as of 22.1.0, the `structlog.threadlocal` module is frozen and will be removed after May 2023.

7.1.1 The `merge_threadlocal` Processor

structlog provides a simple set of functions that allow explicitly binding certain fields to a global (thread-local) context and merge them later using a processor into the event dict.

The general flow of using these functions is:

- Use *structlog.configure()* with *structlog.threadlocal.merge_threadlocal()* as your first processor.

- Call *structlog.threadlocal.clear_threadlocal()* at the beginning of your request handler (or whenever you want to reset the thread-local context).

- Call *structlog.threadlocal.bind_threadlocal()* as an alternative to your bound logger's `bind()` when you want to bind a particular variable to the thread-local context.

- Use *structlog* as normal. Loggers act as they always do, but the *structlog.threadlocal.merge_threadlocal()* processor ensures that any thread-local binds get included in all of your log messages.

- If you want to access the thread-local storage, you use *structlog.threadlocal.get_threadlocal()* and *structlog.threadlocal.get_merged_threadlocal()*.

These functions map 1:1 to the *Context Variables* APIs, so please use those instead:

- *structlog.contextvars.merge_contextvars()*

- *structlog.contextvars.clear_contextvars()*

- *structlog.contextvars.bind_contextvars()*

- *structlog.contextvars.get_contextvars()*

- *structlog.contextvars.get_merged_contextvars()*

7.1.2 Thread-local Contexts

structlog also provides thread-local context storage in a form that you may already know from *Flask* and that makes the *entire context* global to your thread or greenlet.

This makes its behavior more difficult to reason about which is why we generally recommend to use the `merge_contextvars()` route. Therefore, there are currently no plans to re-implement this behavior on top of context variables.

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: `structlog.threadlocal.tmp_bind()`:

```
>>> log.bind(x=42)
<BoundLoggerFilteringAtNotset(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.
>
> - It doesn't play well with `os.fork` and thus `multiprocessing` (unless configured to use the `spawn` start method).

7.1.3 API

structlog.threadlocal.**bind_threadlocal**(*******kw*)

> Put keys and values into the thread-local context.
>
> Use this instead of *bind()* when you want some context to be global (thread-local).
>
> New in version 19.2.0.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**unbind_threadlocal**(**keys*)

> Tries to remove bound *keys* from threadlocal logging context if present.
>
> New in version 20.1.0.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**bound_threadlocal**(***kw*)

> Bind *kw* to the current thread-local context. Unbind or restore *kw* afterwards. Do **not** affect other keys.
>
> Can be used as a context manager or decorator.
>
> New in version 21.4.0.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**get_threadlocal**()

> Return a copy of the current thread-local context.
>
> New in version 21.2.0.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**get_merged_threadlocal**(*bound_logger*)

> Return a copy of the current thread-local context merged with the context from *bound_logger*.
>
> New in version 21.2.0.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**merge_threadlocal**(*logger*, *method_name*, *event_dict*)

> A processor that merges in a global (thread-local) context.
>
> Use this as your first processor in *structlog.configure()* to ensure thread-local context is included in all log calls.
>
> New in version 19.2.0.
>
> Changed in version 20.1.0: This function used to be called `merge_threadlocal_context` and that name is still kept around for backward compatibility.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**clear_threadlocal**()

> Clear the thread-local context.
>
> The typical use-case for this function is to invoke it early in request-handling code.
>
> New in version 19.2.0.
>
> Deprecated since version 22.1.0.

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[Union[Dict[str, Any], Dict[Any, Any]]]`) – Class used for keeping context.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**tmp_bind**(*logger*, ***tmp_values*)

> Bind *tmp_values* to *logger* & memorize current state. Rewind afterwards.
>
> Only works with `structlog.threadlocal.wrap_dict`-based contexts. Use `bound_threadlocal()` for new code.
>
> Deprecated since version 22.1.0.

structlog.threadlocal.**as_immutable**(*logger*)

> Extract the context from a thread local logger into an immutable logger.
>
> > **Parameters**
> > > **logger** (`structlog.typing.BindableLogger`) – A logger with *possibly* thread local state.
> >
> > **Returns**
> > > *BoundLogger* with an immutable context.
> >
> > **Return type**
> > > *TLLogger*
>
> Deprecated since version 22.1.0.

EIGHT

API REFERENCE

8.1 API Reference

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

8.1.1 `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.info("hello", x=42)
y=23 x=42 event='hello'
```

    Parameters

- **args** (*Any*) – *Optional* positional arguments that are passed unmodified to the logger factory. Therefore it depends on the factory what they mean.

- **initial_values** (*Any*) – Values that are used to pre-populate your contexts.

    Returns

        A proxy that creates a correctly configured bound logger when necessary. The type of that bound logger depends on your configuration and is *structlog.BoundLogger* by default.

    Return type

        Any

    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** (*[Any](#)*) – Values that are used to pre-populate your contexts.
> >
> > - **logger_factory_args** (*[Optional[Iterable[Any]]](#)*) – Values that are passed unmodi-
> >   fied as `*logger_factory_args` to the logger factory if not [None](#).
> >
> > **Returns**
> >   A proxy that creates a correctly configured bound logger when necessary.
> >
> > **Return type**
> >   *[Any](#)*
>
> 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*](#) or [*get_logger*](#) are called without arguments.
>
> Can be called several times, keeping an argument at [None](#) leaves it 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** (*[Optional[Iterable[Callable[[Any, str, MutableMapping[str, Any]], Union[Mapping[str, Any], str, bytes, bytearray, Tuple[Any, ...]]]]]](#)*) – The processor chain. See [*Processors*](#) for details.
> >
> > - **wrapper_class** (*[Optional[type[structlog.typing.BindableLogger]]](#)*) – Class to use for wrapping loggers instead of [*structlog.BoundLogger*](#). See [*Standard Library Logging*](#), [*Twisted*](#), and [*Custom Wrappers*](#).
> >
> > - **context_class** (*[Optional[type[Union[Dict[str, Any], Dict[Any, Any]]]]](#)*) – Class to be used for internal context keeping. The default is a [dict](#) and since dictionaries are ordered as of Python 3.6, there's few reasons to change this option.
> >
> > - **logger_factory** (*[Optional[Callable[[...], Any]]](#)*) – Factory to be called to create a new logger that shall be wrapped.
> >
> > - **cache_logger_on_first_use** (*[Optional[bool]](#)*) – [*wrap_logger*](#) doesn't return an ac-
> >   tual 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**(*processors=None*, *wrapper_class=None*, *context_class=None*,
  logger_factory=None, *cache_logger_on_first_use=None*)

> Configures if structlog isn't configured yet.
>
> It does *not* matter whether it 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.

structlog.**get_config**()

> Get a dictionary with the current configuration.
>
> ---
>
> **Note:** Changes to the returned dictionary do *not* affect *structlog*.
>
> ---

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.

> **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.

> **unbind**(**keys*)
>
> > Return a new logger with *keys* removed from the context.
> >
> > > **Raises**
> > >
> > > > `KeyError` – If the key is not part of the context.

---

structlog.**make_filtering_bound_logger**(*min_level*)

> Create a new *FilteringBoundLogger* that only logs *min_level* or higher.
>
> The logger is optimized such that log levels below *min_level* only consist of a `return None`.
>
> All familiar log methods are present, with async variants of each that are prefixed by an `a`. Therefore, the async version of `log.info("hello")` is `await log.ainfo("hello")`.
>
> Additionally it has a `log(self, level: int, **kw: Any)` method to mirror `logging.Logger.log` and *structlog.stdlib.BoundLogger.log*.
>
> Compared to using *structlog*'s standard library integration and the *structlog.stdlib.filter_by_level* processor:
>
> - It's faster because once the logger is built at program start; it's a static class.
>
> - For the same reason you can't change the log level once configured. Use the dynamic approach of *Standard Library Logging* instead, if you need this feature.
>
> - You *can* have (much) more fine-grained filtering by *writing a simple processor*.
>
>> **Parameters**
>>> **min_level** (*int*) – The log level as an integer. You can use the constants from `logging` like `logging.INFO` or pass the values directly. See this table from the logging docs for possible values.
>
> New in version 20.2.0.
>
> Changed in version 21.1.0: The returned loggers are now pickleable.
>
> New in version 20.1.0: The `log()` method.
>
> New in version 22.2.0: Async variants `alog()`, `adebug()`, `ainfo()`, and so forth.

structlog.**get_context**(*bound_logger*)

> Return *bound_logger*'s context.
>
> The type of *bound_logger* and the type returned depend on your configuration.
>
>> **Parameters**
>>> **bound_logger** (*BindableLogger*) – The bound logger whose context you want.
>>
>> **Returns**
>>> The *actual* context from *bound_logger*. It is *not* copied first.
>>
>> **Return type**
>>> *Union*[*Dict*[str, *Any*], *Dict*[*Any*, *Any*]]
>
> New in version 20.2.

class structlog.**PrintLogger**(*file=None*)

> Print events into a file.
>
>> **Parameters**
>>> **file** (*TextIO | None*) – File to print to. (default: `sys.stdout`)

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

> Useful if you follow *current logging best practices*.
>
> Also very useful for testing and examples since `logging` is finicky in doctests.

Changed in version 22.1: The implementation has been switched to use `print` for better monkeypatchability.

critical(*message*)

> Print *message*.

debug(*message*)

> Print *message*.

err(*message*)

> Print *message*.

error(*message*)

> Print *message*.

failure(*message*)

> Print *message*.

fatal(*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** (*TextIO | None*) – File to print to. (default: `sys.stdout`)
>
> Positional arguments are silently ignored.
>
> New in version 0.4.0.

class structlog.**WriteLogger**(*file=None*)

    Write events into a file.

        Parameters

            file (*TextIO | None*) – File to print to. (default: sys.stdout)

```pycon
>>> from structlog import WriteLogger
>>> WriteLogger().info("hello")
hello
```

Useful if you follow *current logging best practices*.

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

A little faster and a little less versatile than *structlog.PrintLogger*.

New in version 22.1.

critical(*message*)

    Write and flush *message*.

debug(*message*)

    Write and flush *message*.

err(*message*)

    Write and flush *message*.

error(*message*)

    Write and flush *message*.

failure(*message*)

    Write and flush *message*.

fatal(*message*)

    Write and flush *message*.

info(*message*)

    Write and flush *message*.

log(*message*)

    Write and flush *message*.

msg(*message*)

    Write and flush *message*.

warning(*message*)

>   Write and flush *message*.

class structlog.**WriteLoggerFactory**(*file=None*)

>   Produce [*WriteLogger*](WriteLogger)s.
>
>   To be used with [`structlog.configure`](structlog.configure)'s `logger_factory`.
>
>   >   **Parameters**
>   >   **file** (`TextIO | None`) – File to print to. (default: `sys.stdout`)
>
>   Positional arguments are silently ignored.
>
>   New in version 22.1.

class structlog.**BytesLogger**(*file=None*)

>   Writes bytes into a file.
>
>   >   **Parameters**
>   >   **file** (`BinaryIO | None`) – File to print to. (default: `sys.stdout.buffer`)
>
>   Useful if you follow *current logging best practices* together with a formatter that returns bytes (e.g. orjson).
>
>   New in version 20.2.0.
>
>   **critical**(*message*)
>
>   >   Write *message*.
>
>   **debug**(*message*)
>
>   >   Write *message*.
>
>   **err**(*message*)
>
>   >   Write *message*.
>
>   **error**(*message*)
>
>   >   Write *message*.
>
>   **failure**(*message*)
>
>   >   Write *message*.
>
>   **fatal**(*message*)
>
>   >   Write *message*.
>
>   **info**(*message*)
>
>   >   Write *message*.

log(*message*)

> Write *message*.

msg(*message*)

> Write *message*.

warning(*message*)

> Write *message*.

class structlog.**BytesLoggerFactory**(*file=None*)

> Produce [BytesLogger](#)s.
>
> To be used with [structlog.configure](#)'s logger_factory.
>
> > **Parameters**
> >
> > **file** (*BinaryIO | None*) – File to print to. (default: sys.stdout.buffer)
>
> Positional arguments are silently ignored.
>
> New in version 20.2.0.

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.stdlib.BoundLogger](#).
> - [structlog.twisted.BoundLogger](#),
>
> See also [Custom Wrappers](#).

_logger: Any

> 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** ([str](#) | *None*) – The event – usually the first positional argument to a logger.

- **event_kw**(*dict[str, Any]*) – Additional event keywords. For example if someone calls `log.info("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 str, bytes, bytearray, tuple, or a dict.
>
> **Returns**
> > `tuple` of (*args, **kw)
>
> **Return type**
> > tuple[*Sequence*[*Any*], *Mapping*[str, *Any*]]

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`.

Changed in version 20.2.0: Allow final processor to return `bytes`.

Changed in version 21.2.0: Allow final processor to return a `bytearray`.

_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** (*Optional[str]*) – The event – usually the first positional argument to a logger.
> >
> > - **event_kw** (*Any*) – Additional event keywords. For example if someone calls `log.info("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.

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.

> **try_unbind**(**keys*)
>
>> Like *unbind()*, but best effort: missing keys are ignored.
>>
>> New in version 18.2.0.

> **unbind**(**keys*)
>
>> Return a new logger with *keys* removed from the context.
>>
>>> **Raises**
>>>> **KeyError** – If the key is not part of the context.

8.1.2 `structlog.dev` Module

Helpers that make development with *structlog* more pleasant.

See also the narrative documentation in `Development Tools`.

class `structlog.dev.ConsoleRenderer`(*pad_event=30*, *colors=True*, *force_colors=False*, *repr_native_str=False*, *level_styles=None*, *exception_formatter=<function plain_traceback>*, *sort_keys=True*, *event_key='event'*)

> Render `event_dict` nicely aligned, possibly in colors, and ordered.
>
> If `event_dict` contains a true-ish `exc_info` key, it will be rendered *after* the log line. If Rich or better-exceptions are present, in colors and with extra context.
>
>> **Parameters**
>>
>> - **pad_event** (*int*) – Pad the event to this many characters.
>>
>> - **colors** (*bool*) – Use colors for a nicer output. `True` by default. On Windows only if Colorama is installed.
>>
>> - **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. Only meaningful on Windows.
>>
>> - **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** (*Styles | None*) – 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*
>>
>> - **exception_formatter** (*ExceptionRenderer*) – A callable to render `exc_infos`. If rich or better-exceptions are installed, they are used for pretty-printing by default (rich taking precedence). You can also manually set it to *plain_traceback*, *better_traceback*, *rich_traceback*, or implement your own.
>>
>> - **sort_keys** (*bool*) – Whether to sort keys when formatting. `True` by default.
>>
>> - **event_key** (*str*) – The key to look for the main log message. Needed when you rename it e.g. using *structlog.processors.EventRenamer*.
>
> Requires the Colorama package if *colors* is `True` **on Windows**.
>
> 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*

Changed in version 19.2: *Colorama* now initializes lazily to avoid unwanted initializations as `ConsoleRenderer` is used by default.

Changed in version 19.2: Can be pickled now.

Changed in version 20.1: *Colorama* does not initialize lazily on Windows anymore because it breaks rendering.

Changed in version 21.1: It is additionally possible to set the logger name using the `logger_name` key in the `event_dict`.

New in version 21.2: *exception_formatter*

Changed in version 21.2: `ConsoleRenderer` now handles the `exc_info` event dict key itself. Do **not** use the `structlog.processors.format_exc_info` processor together with `ConsoleRenderer` anymore! It will keep working, but you can't have customize exception formatting and a warning will be raised if you ask for it.

Changed in version 21.2: The colors keyword now defaults to True on non-Windows systems, and either True or False in Windows depending on whether *Colorama* is installed.

New in version 21.3.0: *sort_keys*

New in version 22.1: *event_key*

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`.

structlog.dev.**plain_traceback**(*sio*, *exc_info*)

"Pretty"-print *exc_info* to *sio* using our own plain formatter.

To be passed into `ConsoleRenderer`'s `exception_formatter` argument.

Used by default if neither *Rich* not *better-exceptions* are present.

New in version 21.2.

structlog.dev.**rich_traceback**(*sio*, *exc_info*)

Pretty-print *exc_info* to *sio* using the *Rich* package.

To be passed into `ConsoleRenderer`'s `exception_formatter` argument.

Used by default if *Rich* is installed.

New in version 21.2.

structlog.dev.**better_traceback**(*sio*, *exc_info*)

>   Pretty-print *exc_info* to *sio* using the *better-exceptions* package.

>   To be passed into *ConsoleRenderer*'s `exception_formatter` argument.

>   Used by default if *better-exceptions* is installed and *Rich* is absent.

>   New in version 21.2.

structlog.dev.**set_exc_info**(*logger*, *method_name*, *event_dict*)

>   Set `event_dict["exc_info"]` = `True` if *method_name* is `"exception"`.

>   Do nothing if the name is different or `exc_info` is already set.

8.1.3 `structlog.testing` Module

Helpers to test your application's logging behavior.

New in version 20.1.0.

See *Testing*.

structlog.testing.**capture_logs**()

>   Context manager that appends all logging statements to its yielded list while it is active. Disables all configured processors for the duration of the context manager.

>   Attention: this is **not** thread-safe!

>   New in version 20.1.0.

class structlog.testing.**LogCapture**

>   Class for capturing log messages in its entries list. Generally you should use *structlog.testing.capture_logs*, but you can use this class if you want to capture logs with other patterns.

>   > **Variables**
>   >   **entries** (`List[structlog.typing.EventDict]`) – The captured log entries.

>   New in version 20.1.0.

class structlog.testing.**CapturingLogger**

>   Store the method calls that it's been called with.

>   This is nicer than *ReturnLogger* for unit tests because the bound logger doesn't have to cooperate.

>   **Any** method name is supported.

>   New in version 20.2.0.

```
>>> from pprint import pprint
>>> cl = structlog.testing.CapturingLogger()
>>> cl.info("hello")
>>> cl.info("hello", when="again")
>>> pprint(cl.calls)
[CapturedCall(method_name='info', args=('hello',), kwargs={}),
 CapturedCall(method_name='info', args=('hello',), kwargs={'when': 'again'})]
```

class structlog.testing.**CapturingLoggerFactory**

Produce and cache *CapturingLogger*s.

Each factory produces and re-uses only **one** logger.

You can access it via the `logger` attribute.

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

Positional arguments are silently ignored.

New in version 20.2.0.

class structlog.testing.**CapturedCall**(*method_name*, *args*, *kwargs*)

A call as captured by *CapturingLogger*.

Can also be unpacked like a tuple.

> **Parameters**
>
> - **method_name** (*str*) – The method name that got called.
>
> - **args** (*tuple[Any, ...]*) – A tuple of the positional arguments.
>
> - **kwargs** (*dict[str, Any]*) – A dict of the keyword arguments.

New in version 20.2.0.

class structlog.testing.**ReturnLogger**

Return the arguments that it's called with.

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

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

fatal(*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.testing.`**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.

8.1.4 `structlog.contextvars` Module

Primitives to deal with a concurrency supporting context, as introduced in Python 3.7 as `contextvars`.

New in version 20.1.0.

Changed in version 21.1.0: Reimplemented without using a single dict as context carrier for improved isolation. Every key-value pair is a separate `contextvars.ContextVar` now.

See *Context Variables*.

`structlog.contextvars.`**bind_contextvars**(***kw*)

> Put keys and values into the context-local context.
>
> Use this instead of *bind()* when you want some context to be global (context-local).
>
> Return the mapping of `contextvars.Token`s resulting from setting the backing `ContextVar`s. Suitable for passing to *reset_contextvars()*.
>
> New in version 20.1.0.
>
> Changed in version 21.1.0: Return the `contextvars.Token` mapping rather than None. See also the toplevel note.

structlog.contextvars.**bound_contextvars**(***kw*)

> Bind *kw* to the current context-local context. Unbind or restore *kw* afterwards. Do **not** affect other keys.
>
> Can be used as a context manager or decorator.
>
> New in version 21.4.0.

structlog.contextvars.**get_contextvars**()

> Return a copy of the *structlog*-specific context-local context.
>
> New in version 21.2.0.

structlog.contextvars.**get_merged_contextvars**(*bound_logger*)

> Return a copy of the current context-local context merged with the context from *bound_logger*.
>
> New in version 21.2.0.

structlog.contextvars.**merge_contextvars**(*logger*, *method_name*, *event_dict*)

> A processor that merges in a global (context-local) context.
>
> Use this as your first processor in `structlog.configure()` to ensure context-local context is included in all log calls.
>
> New in version 20.1.0.
>
> Changed in version 21.1.0: See toplevel note.

structlog.contextvars.**clear_contextvars**()

> Clear the context-local context.
>
> The typical use-case for this function is to invoke it early in request- handling code.
>
> New in version 20.1.0.
>
> Changed in version 21.1.0: See toplevel note.

structlog.contextvars.**unbind_contextvars**(**keys*)

> Remove *keys* from the context-local context if they are present.
>
> Use this instead of `unbind()` when you want to remove keys from a global (context-local) context.
>
> New in version 20.1.0.
>
> Changed in version 21.1.0: See toplevel note.

structlog.contextvars.**reset_contextvars**(***kw*)

> Reset contextvars corresponding to the given Tokens.
>
> New in version 21.1.0.

8.1.5 `structlog.threadlocal` Module

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

See *Legacy Thread-local Context*, but please use *Context Variables* instead.

Deprecated since version 22.1.0.

8.1.6 `structlog.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, **dumps_kw)`.
>
> > **Parameters**
> >
> > - **dumps_kw** (*Any*) – Are passed unmodified to *serializer*. If *default* is passed, it will disable support for `__structlog__`-based serialization.
> > - **serializer** (*Callable[..., str | bytes]*) – A `json.dumps()`-compatible callable that will be used to format the string. This can be used to use alternative JSON encoders like orjson or RapidJSON (default: `json.dumps()`).
>
> New in version 0.2.0: Support for `__structlog__` serialization method.
>
> New in version 15.4.0: *serializer* parameter.
>
> New in version 18.2.0: Serializer's *default* parameter can be overwritten now.

```
>>> 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:
...     def __structlog__(self):
...         return ["C1!"]
...     def __repr__(self):
...         return "__structlog__ took precedence"
>>> class C2:
...     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.
>
> If you choose to pass a *default* parameter as part of *json_kw*, support for `__structlog__` is disabled. This can be useful when used together with more elegant serialization methods like `functools.singledispatch()`: Better Python Object Serialization.
>
> ---
>
> **Tip:** If you use this processor, you may also wish to add structured tracebacks for exceptions. You can do this by adding the `dict_tracebacks` to your list of processors:

```
>>> structlog.configure(
...     processors=[
...         structlog.processors.dict_tracebacks,
...         structlog.processors.JSONRenderer(),
...     ],
... )
>>> log = structlog.get_logger()
>>> var = "spam"
>>> try:
...     1 / 0
... except ZeroDivisionError:
...     log.exception("Cannot compute!")
{"event": "Cannot compute!", "exception": [{"exc_type": "ZeroDivisionError", "exc_
→value": "division by zero", "syntax_error": null, "is_cause": false, "frames": [{
→"filename": "<doctest default[3]>", "lineno": 2, "name": "<module>", "line": "",
→"locals": {..., "var": "spam"}}]}]}
```

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** (*Sequence[str] | None*) – 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. 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.**LogfmtRenderer**(*sort_keys=False*, *key_order=None*, *drop_missing=False*, *bool_as_flag=True*)

Render event_dict using the logfmt format.

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

- **key_order** (*Sequence[str] | None*) – List of keys that should be rendered in this exact order. Missing keys are rendered with empty values, 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 with empty values.

- **bool_as_flag** (*bool*) – When True, render {"flag": True} as flag, instead of flag=true. {"flag": False} is always rendered as flag=false.

Raises
    ValueError – If a key contains non printable or space characters.

New in version 21.5.0.

```
>>> from structlog.processors import LogfmtRenderer
>>> event_dict = {"a": 42, "b": [1, 2, 3], "flag": True}
>>> LogfmtRenderer(sort_keys=True)(None, None, event_dict)
'a=42 b="[1, 2, 3]" flag'
>>> LogfmtRenderer(key_order=["b", "a"], bool_as_flag=False)(None, None, event_dict)
'b="[1, 2, 3]" a=42 flag=true'
```

class structlog.processors.**EventRenamer**(*to*, *replace_by=None*)

Rename the event key in event dicts.

This is useful if you want to use consistent log message keys across platforms and/or use the event key for something custom.

> **Warning:** It's recommended to put this processor right before the renderer, since some processors may rely on the presence and meaning of the event key.

Parameters

- **to** (*str*) – Rename event_dict["event"] to event_dict[to]

- **replace_by** (*str | None*) – Rename event_dict[replace_by] to event_dict["event"]. *replace_by* missing from event_dict is handled gracefully.

New in version 22.1.

See also the *Renaming the event Key* recipe.

structlog.processors.**add_log_level**(*logger*, *method_name*, *event_dict*)

Add the log level to the event dict under the level key.

Since that's just the log method name, this processor works with non-stdlib logging as well. Therefore it's importable both from structlog.processors as well as from structlog.stdlib.

New in version 15.0.0.

Changed in version 20.2.0: Importable from structlog.processors (additionally to structlog.stdlib).

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.

class structlog.processors.**ExceptionRenderer**(*exception_formatter=<function _format_exception>*)

> Replace an `exc_info` field with an `exception` field which is rendered by *exception_formatter*.
>
> The contents of the `exception` field depends on the return value of the [*ExceptionTransformer*](#) that is used:
>
> - The default produces a formatted string via Python's built-in traceback formatting.
>
> - The [*ExceptionDictTransformer*](#) a list of stack dicts that can be serialized to JSON.
>
> If *event_dict* contains the key `exc_info`, there are three possible behaviors:
>
> 1. If the value is a tuple, render it into the key `exception`.
>
> 2. If the value is an Exception render it into the key `exception`.
>
> 3. 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.
>
> > **Parameters**
> >
> > **exception_formatter** ([ExceptionTransformer](#)) – A callable that is used to format the exception from the `exc_info` field.
>
> New in version 22.1.

structlog.processors.**format_exc_info**(*logger*, *name*, *event_dict*)

> Replace an `exc_info` field with an `exception` string field using Python's built-in traceback formatting.
>
> If *event_dict* contains the key `exc_info`, there are three possible behaviors:
>
> 1. If the value is a tuple, render it into the key `exception`.
>
> 2. If the value is an Exception render it into the key `exception`.
>
> 3. If the value is 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):...
```

structlog.processors.**dict_tracebacks**(*logger*, *name*, *event_dict*)

> Replace an exc_info field with an exception field containing structured tracebacks suitable for, e.g., JSON output.
>
> It is a shortcut for *ExceptionRenderer* with a *ExceptionDictTransformer*.
>
> The treatment of the exc_info key is identical to *format_exc_info*.
>
> New in version 22.1.

```
>>> from structlog.processors import dict_tracebacks
>>> try:
...     raise ValueError("onoes")
... except ValueError:
...     dict_tracebacks(None, None, {"exc_info": True})
{'exception': [{'exc_type': 'ValueError', 'exc_value': 'onoes', ..., 'frames': [{
↪'filename': ...
```

class structlog.processors.**StackInfoRenderer**(*additional_ignores=None*)

> 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 and works analogously to the *stack_info* argument of the Python standard library logging.
>
> > **Parameters**
> >   **additional_ignores** (*list[str] | None*) – By default, stack frames coming from *structlog* are ignored. With this argument you can add additional names that are ignored, before the stack starts being rendered. They are matched using startswith(), so they don't have to match exactly. The names are used to find the first relevant name, therefore once a frame is found that doesn't start with *structlog* or one of *additional_ignores*, **no filtering** is applied to subsequent frames.
>
> New in version 0.4.0.
>
> New in version 22.1.0: *additional_ignores*

class structlog.processors.**ExceptionPrettyPrinter**(*file=None*, *exception_formatter=<function _format_exception>*)

> Pretty print exceptions and remove them from the event_dict.
>
> > **Parameters**
> >   **file** (*TextIO | None*) – 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*, *key='timestamp'*)

    Add a timestamp to `event_dict`.

        Parameters

- **fmt** (`str` | *None*) – 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.

    Changed in version 19.2: Can be pickled now.

```
>>> 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'}
```

class structlog.processors.**CallsiteParameter**(*value*)

    Callsite parameters that can be added to an event dictionary with the *structlog.processors.CallsiteParameterAdder* processor class.

    The string values of the members of this enum will be used as the keys for the callsite parameters in the event dictionary.

    New in version 21.5.0.

    FILENAME = 'filename'

        The basename part of the full path to the python source file of the callsite.

    FUNC_NAME = 'func_name'

        The name of the function that the callsite was in.

    LINENO = 'lineno'

        The line number of the callsite.

    MODULE = 'module'

        The python module the callsite was in. This mimics the module attribute of `logging.LogRecord` objects and will be the basename, without extension, of the full path to the python source file of the callsite.

    PATHNAME = 'pathname'

        The full path to the python source file of the callsite.

    PROCESS = 'process'

        The ID of the process the callsite was executed in.

    PROCESS_NAME = 'process_name'

        The name of the process the callsite was executed in.

    THREAD = 'thread'

        The ID of the thread the callsite was executed in.

THREAD_NAME = 'thread_name'
> The name of the thread the callsite was executed in.

class structlog.processors.**CallsiteParameterAdder**(*parameters={<CallsiteParameter.THREAD_NAME: 'thread_name'>, <CallsiteParameter.PROCESS: 'process'>, <CallsiteParameter.MODULE: 'module'>, <CallsiteParameter.THREAD: 'thread'>, <CallsiteParameter.PROCESS_NAME: 'process_name'>, <CallsiteParameter.LINENO: 'lineno'>, <CallsiteParameter.FILENAME: 'filename'>, <CallsiteParameter.PATHNAME: 'pathname'>, <CallsiteParameter.FUNC_NAME: 'func_name'>}, additional_ignores=None*)

Adds parameters of the callsite that an event dictionary originated from to the event dictionary. This processor can be used to enrich events dictionaries with information such as the function name, line number and filename that an event dictionary originated from.

> **Warning:** This processor cannot detect the correct callsite for invocation of async functions.

If the event dictionary has an embedded logging.LogRecord object and did not originate from *structlog* then the callsite information will be determined from the logging.LogRecord object. For event dictionaries without an embedded logging.LogRecord object the callsite will be determined from the stack trace, ignoring all intra-structlog calls, calls from the logging module, and stack frames from modules with names that start with values in additional_ignores, if it is specified.

The keys used for callsite parameters in the event dictionary are the string values of *CallsiteParameter* enum members.

> **Parameters**
> - **parameters** (*Collection[CallsiteParameter]*) – A collection of *CallsiteParameter* values that should be added to the event dictionary.
> - **additional_ignores** (*list[str] | None*) – Additional names with which a stack frame's module name must not start for it to be considered when determening the callsite.

> **Note:** When used with *structlog.stdlib.ProcessorFormatter* the most efficient configuration is to either use this processor in foreign_pre_chain of *structlog.stdlib.ProcessorFormatter* and in processors of *structlog.configure*, or to use it in processors of *structlog.stdlib.ProcessorFormatter* without using it in processors of *structlog.configure* and foreign_pre_chain of *structlog.stdlib.ProcessorFormatter*.

New in version 21.5.0.

8.1.7 `structlog.stdlib` Module

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

See also *structlog's standard library support*.

`structlog.stdlib.`**`recreate_defaults`**(*, *log_level=0*)

> Recreate defaults on top of standard library's logging.
>
> The output looks the same, but goes through `logging`.
>
> As with vanilla defaults, the backwards-compatibility guarantees don't apply to the settings applied here.
>
> > **Parameters**
> > > **log_level** (`int` | *None*) – If `None`, don't configure standard library logging **at all**.
> > >
> > > Otherwise configure it to log to `sys.stdout` at *log_level* (`logging.NOTSET` being the default).
> > >
> > > If you need more control over `logging`, pass `None` here and configure it yourself.
>
> New in version 22.1.

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

> Only calls *structlog.get_logger*, but has the correct type hints.
>
> > **Warning:** Does **not** check whether – or ensure that – you've configured *structlog* for standard library `logging`!
> >
> > See *Standard Library Logging* for details.
>
> New in version 20.2.0.

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,
)
```

> It also contains a bunch of properties that pass-through to the wrapped `logging.Logger` which should make it work as a drop-in replacement.
>
> **bind**(***new_values*)
>
> > Return a new logger with *new_values* added to the existing ones.
>
> **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=None*, **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.

try_unbind(**keys*)

> Like *unbind()*, but best effort: missing keys are ignored.
>
> New in version 18.2.0.

unbind(**keys*)

> Return a new logger with *keys* removed from the context.
>
> > **Raises**
> > > **KeyError** – If the key is not part of the context.

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.**AsyncBoundLogger**(*logger*, *processors*, *context*, ***, *_sync_bl=None*, *_loop=None*)

Wraps a *BoundLogger* & exposes its logging methods as `async` versions.

Instead of blocking the program, they are run asynchronously in a thread pool executor.

This means more computational overhead per log call. But it also means that the processor chain (e.g. JSON serialization) and I/O won't block your whole application.

Only available for Python 3.7 and later.

> **Variables**
>> **sync_bl** (`structlog.stdlib.BoundLogger`) – The wrapped synchronous logger. It is useful to be able to log synchronously occasionally.

New in version 20.2.0.

Changed in version 20.2.0: fix _dispatch_to_sync contextvars usage

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[str] | None`) – 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"]`. This argument is called *additional_ignores* in other APIs throughout *structlog*.
>
> **__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.
>>
>> 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**(*_, __, 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.
>
> Changed in version 22.1.0: `exc_info`, `stack_info`, and `stackLevel` are passed as proper kwargs and not put into `extra`.

structlog.stdlib.**filter_by_level**(*logger*, *method_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 under the `level` key.

   Since that's just the log method name, this processor works with non-stdlib logging as well. Therefore it's importable both from *structlog.processors* as well as from *structlog.stdlib*.

   New in version 15.0.0.

   Changed in version 20.2.0: Importable from *structlog.processors* (additionally to *structlog.stdlib*).

structlog.stdlib.**add_log_level_number**(*logger*, *method_name*, *event_dict*)

   Add the log level number to the event dict.

   Log level numbers map to the log level names. The Python stdlib uses them for filtering logic. This adds the same numbers so users can leverage similar filtering. Compare:

```
level in ("warning", "error", "critical")
level_number >= 30
```

   The mapping of names to numbers is in `structlog.stdlib._log_levels._NAME_TO_LEVEL`.

   New in version 18.2.0.

structlog.stdlib.**add_logger_name**(*logger*, *method_name*, *event_dict*)

   Add the logger name to the event dict.

structlog.stdlib.**ExtraAdder**(*allow=None*)

   Add extra attributes of `logging.LogRecord` objects to the event dictionary.

   This processor can be used for adding data passed in the `extra` parameter of the `logging` module's log methods to the event dictionary.

   > **Parameters**
   >   **allow** (*Collection[str] | None*) – An optional collection of attributes that, if present in `logging.LogRecord` objects, will be copied to event dictionaries.
   >
   >   If `allow` is None all attributes of `logging.LogRecord` objects that do not exist on a standard `logging.LogRecord` object will be copied to event dictionaries.

   New in version 21.5.0.

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=None*, *processors=()*, *foreign_pre_chain=None*, *keep_exc_info=False*, *keep_stack_info=False*, *logger=None*, *pass_foreign_args=False*, **args*, ***kwargs*)

Call *structlog* processors on logging.LogRecords.

This is an implementation of a logging.Formatter that can be used to format log entries from both *structlog* and logging.

Its static method *wrap_for_formatter* must be the final processor in *structlog*'s processor chain.

Please refer to *Rendering Using structlog-based Formatters Within logging* for examples.

> **Parameters**
>
> - **foreign_pre_chain** (*Sequence[Processor] | None*) – If not None, it is used as a processor chain that is applied to **non**-*structlog* log entries before the event dictionary is passed to *processors*. (default: None)
>
> - **processors** (*Sequence[Processor] | None*) – A chain of *structlog* processors that is used to process **all** log entries. The last one must render to a str which then gets passed on to logging for output.
>
>   Compared to *structlog*'s regular processor chains, there's a few differences:
>
>   – The event dictionary contains two additional keys:
>
>   1. _record: a logging.LogRecord that either was created using logging APIs, **or** is a wrapped *structlog* log entry created by *wrap_for_formatter*.
>
>   2. _from_structlog: a bool that indicates whether or not _record was created by a *structlog* logger.
>
>   Since you most likely don't want _record and _from_structlog in your log files, we've added the static method *remove_processors_meta* to ProcessorFormatter that you can add just before your renderer.
>
>   – Since this is a logging *formatter*, raising *structlog.DropEvent* will crash your application.
>
> - **keep_exc_info** (*bool*) – exc_info on logging.LogRecords 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 stack_info. (default: False)

- **logger** (*logging.Logger* | *None*) – Logger which we want to push through the *structlog* processor chain. This parameter is necessary for some of the processors like *filter_by_level*. (default: None)

- **pass_foreign_args** (*bool*) – If True, pass a foreign log record's `args` attribute to the `event_dict` under `positional_args` key. (default: False)

- **processor** (*Processor* | *None*) – A single *structlog* processor used for rendering the event dictionary before passing it off to `logging`. Must return a `str`. The event dictionary does **not** contain `_record` and `_from_structlog`.

  This parameter exists for historic reasons. Please consider using *processors* instead.

  Raises
  > **TypeError** – If both or neither *processor* and *processors* are passed.

New in version 17.1.0.

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

New in version 19.2.0: *logger*

New in version 19.2.0: *pass_foreign_args*

New in version 21.3.0: *processors*

Deprecated since version 21.3.0: *processor* (singular) in favor of *processors* (plural). Removal is not planned.

static **remove_processors_meta**(_, __, *event_dict*)

  Remove `_record` and `_from_structlog` from *event_dict*.

  These keys are added to the event dictionary, before `ProcessorFormatter`'s *processors* are run.

  New in version 21.3.0.

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.

8.1.8 `structlog.tracebacks` Module

Extract a structured traceback from an exception.

Contributed by Will McGugan (see https://github.com/hynek/structlog/pull/407#issuecomment-1150926246) from Rich: https://github.com/Textualize/rich/blob/972dedff/rich/traceback.py

structlog.tracebacks.**extract**(*exc_type*, *exc_value*, *traceback*, *, *show_locals=False*, *locals_max_string=80*)

  Extract traceback information.

  Parameters

  - **exc_type** (*type[BaseException]*) – Exception type.

  - **exc_value** (*BaseException*) – Exception value.

  - **traceback** (*types.TracebackType* | *None*) – Python Traceback object.

- **show_locals** (*bool*) – Enable display of local variables. Defaults to False.
- **locals_max_string** (*int*) – Maximum length of string before truncating, or None to disable.
- **max_frames** – Maximum number of frames in each stack

> **Returns**
> A Trace instance with structured information about all exceptions.
>
> **Return type**
> Trace

New in version 22.1.

class structlog.tracebacks.**ExceptionDictTransformer**(*show_locals=True*, *locals_max_string=80*, *max_frames=50*)

Return a list of exception stack dictionaries for for an exception.

These dictionaries are based on *Stack* instances generated by *extract()* and can be dumped to JSON.

> **Parameters**
> - **show_locals** (*bool*) – Whether or not to include the values of a stack frame's local variables.
> - **locals_max_string** (*int*) – The maximum length after which long string representations are truncated.
> - **max_frames** (*int*) – Maximum number of frames in each stack. Frames are removed from the inside out. The idea is, that the first frames represent your code responsible for the exception and last frames the code where the exception actually happened. With larger web frameworks, this does not always work, so you should stick with the default.

class structlog.tracebacks.**Trace**(*stacks*)

Container for a list of stack traces.

class structlog.tracebacks.**Stack**(*exc_type*, *exc_value*, *syntax_error=None*, *is_cause=False*, *frames=<factory>*)

Represents an exception and a list of stack frames.

class structlog.tracebacks.**Frame**(*filename*, *lineno*, *name*, *line=''*, *locals=None*)

Represents a single stack frame.

class structlog.tracebacks.**SyntaxError_**(*offset*, *filename*, *line*, *lineno*, *msg*)

Contains detailed information about SyntaxError exceptions.

8.1.9 `structlog.typing` Module

Type information used throughout *structlog*.

For now, they are considered provisional. Especially *BindableLogger* will probably change to something more elegant.

New in version 22.2.

class `structlog.typing.`**`BindableLogger`**(**args*, ***kwargs*)

> **Protocol**: Methods shared among all bound loggers and that are relied on by *structlog*.
>
> New in version 20.2.
>
> Additionally to the methods listed below, bound loggers **must** have a `__init__` method with the following signature:
>
> **`__init__`**(*self*, *wrapped_logger: WrappedLogger*, *processors: Iterable[Processor]*, *context: Context*) →
> > None
>
> Unfortunately it's impossible to define initializers using **PEP 544** Protocols.
>
> They currently also have to carry a *Context* as a `_context` attribute.
>
> ---
>
> **Note:** Currently Sphinx has no support for Protocols, so please click `[source]` for this entry to see the full definition.
>
> ---

class `structlog.typing.`**`FilteringBoundLogger`**(**args*, ***kwargs*)

> **Protocol**: A *BindableLogger* that filters by a level.
>
> The only way to instantiate one is using *make_filtering_bound_logger*.
>
> New in version 20.2.0.
>
> New in version 22.2.0: String interpolation using positional arguments.
>
> New in version 22.2.0: Async variants `alog()`, `adebug()`, `ainfo()`, and so forth.
>
> Changed in version 22.3.0: String interpolation is only attempted if positional arguments are passed.
>
> ---
>
> **Note:** Currently Sphinx has no support for Protocols, so please click `[source]` for this entry to see the full definition.
>
> ---

class `structlog.typing.`**`ExceptionTransformer`**(**args*, ***kwargs*)

> **Protocol:** A callable that transforms an *ExcInfo* into another datastructure.
>
> The result should be something that your renderer can work with, e.g., a `str` or a JSON-serializable `dict`.
>
> Used by *structlog.processors.format_exc_info()* and *structlog.processors.ExceptionPrettyPrinter*.
>
> > **Parameters**
> > > **exc_info** – Is the exception tuple to format
> >
> > **Returns**
> > > Anything that can be rendered by the last processor in your chain, e.g., a string or a JSON-serializable structure.

New in version 22.1.

Note: Currently Sphinx has no support for Protocols, so please click [source] for this entry to see the full definition.

---

structlog.typing.**EventDict**

> An event dictionary as it is passed into processors.
>
> It's created by copying the configured *Context* but doesn't need to support copy itself.
>
> New in version 20.2.
>
> alias of MutableMapping[str, Any]

structlog.typing.**WrappedLogger = typing.Any**

> A logger that is wrapped by a bound logger and is ultimately responsible for the output of the log entries.
>
> *structlog* makes *no* assumptions about it.
>
> New in version 20.2.

structlog.typing.**Processor**

> A callable that is part of the processor chain.
>
> See *Processors*.
>
> New in version 20.2.
>
> alias of Callable[[Any, str, MutableMapping[str, Any]], Union[Mapping[str, Any], str, bytes, bytearray, Tuple[Any, ...]]]

structlog.typing.**Context**

> A dict-like context carrier.
>
> New in version 20.2.
>
> alias of Union[Dict[str, Any], Dict[Any, Any]]

structlog.typing.**ExcInfo**

> An exception info tuple as returned by sys.exc_info.
>
> New in version 20.2.
>
> alias of Tuple[Type[BaseException], BaseException, Optional[TracebackType]]

structlog.typing.**ExceptionRenderer**

> A callable that pretty-prints an *ExcInfo* into a file-like object.
>
> Used by *structlog.dev.ConsoleRenderer*.
>
> New in version 21.2.
>
> alias of Callable[[TextIO, Tuple[Type[BaseException], BaseException, Optional[TracebackType]]], None]

8.1.10 `structlog.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.

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.

unbind(**keys*)

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

> **Raises**
> **KeyError** – If the key is not part of the context.

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[[WrappedLogger, str, EventDict], str] | None`) – 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. Also does *not* 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** (*TextIO*) – File to print to.
>
> New in version 0.2.0.

PROJECT INFORMATION

- **License**: *dual* Apache License, version 2 **and** MIT
- **Get Help**: please use the *structlog* tag on *Stack Overflow*
- **Supported Python Versions**: 3.7 and later
- **PyPI**
- **Source Code**
- **Documentation**
- **Changelog**
- **Third-party Extensions**

9.1 *structlog* for Enterprise

Available as part of the Tidelift Subscription.

The maintainers of *structlog* and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source packages you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact packages you use. Learn more.

9.2 License and Hall of Fame

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

Any contribution intentionally submitted for inclusion in the work by you, as defined in the Apache-2.0 license, shall be dual licensed as above, without any additional terms or conditions.

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` project.

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

- Apache License 2.0
- MIT

9.2.1 Credits

structlog is written and maintained by Hynek Schlawack. The idea of bound loggers is inspired by previous work by Jean-Paul Calderone and David Reid.

The development is kindly supported by my employer Variomedia AG, *structlog*'s Tidelift subscribers, and all my amazing GitHub Sponsors.

A full list of contributors can be found in GitHub's overview.

The logs-loving futuristic beaver logo has been contributed by Russell Keith-Magee.

9.3 Changelog

All notable changes to this project will be documented in this file.

The format is based on *Keep a Changelog* and this project adheres to *Calendar Versioning*.

The **first number** of the version is the year. The **second number** is incremented with each release, starting at 1 for each year. The **third number** is for emergencies when we need to start branches for older releases.

You can find out backwards-compatibility policy here.

9.3.1 22.3.0 - 2022-11-24

Changed

- String interpolation in `FilteringBoundLogger` (used by default) is now only attempted if positional arguments are passed. This prevents crashes if something different than a string is passed for the *event* argument. #475

Fixed

- String interpolation doesn't cause crashes in filtered log call anymore. #478

9.3.2 22.2.0 - 2022-11-19

Deprecated

- Accessing package metadata as attributes on the *structlog* module is deprecated (e.g. `structlog.__version__`). Please use `importlib.metadata` instead (for Python 3.7: the *importlib-metadata* PyPI package).

- The `structlog.types` module is now deprecated in favor of the `structlog.typing` module. It seems like the Python typing community is settling on this name.

Added

- `FilteringBoundLogger` (used by default) now allows for string interpolation using positional arguments:

```
>>> log.info("Hello %s! The answer is %d.", "World", 42, x=1)
2022-10-07 10:04.31 [info     ] Hello World! The answer is 42. x=1
```

  #454

- `FilteringBoundLogger` now also has support for *asyncio*-based logging. Instead of a wrapper class like `structlog.stdlib.AsyncBoundLogger`, async equivalents have been added for all logging methods. So instead of `log.info("hello")` you can also write `await log.ainfo("hello")` in async functions and methods.

  This seems like the better approach and if it's liked by the community, `structlog.stdlib.BoundLogger` will get those methods too. #457

Changed

- The documentation has been **heavily** overhauled. Have a look if you haven't lately! Especially the graphs in the standard library chapter have proven valuable to many.

- The build backend has been switched to *Hatch*.

Fixed

- The timestamps in the default configuration now use the correct separator (:) for seconds.

9.3.3 22.1.0 - 2022-07-20

Removed

- Python 3.6 is not supported anymore.

- Pickling is now only possible with protocol version 3 and newer.

Deprecated

- The entire `structlog.threadlocal` module is deprecated. Please use the primitives from `structlog.contextvars` instead.

  If you're using the modern APIs (`bind_threadlocal()` / `merge_threadlocal()`) it's enough to replace them 1:1 with their `contextvars` counterparts. The old approach around `wrap_dict()` has been discouraged for a while.

  Currently there are no concrete plans to remove the module, but no patches against it will be accepted from now on. #409

Added

- `structlog.processors.StackInfoRenderer` now has an *additional_ignores* parameter that allows you to filter out your own logging layer. #396

- Added `structlog.WriteLogger`, a faster – but more low-level – alternative to `structlog.PrintLogger`. It works the way `PrintLogger` used to work in previous versions. #403 #404

- `structlog.make_filtering_bound_logger()`-returned loggers now also have a `log()` method to match the `structlog.stdlib.BoundLogger` signature closer. #413

- Added structured logging of tracebacks via the `structlog.tracebacks` module, and most notably the `structlog.tracebacks.ExceptionDictTransformer` which can be used with the new `structlog.processors.ExceptionRenderer` to render JSON tracebacks. #407

- `structlog.stdlib.recreate_defaults(log_level=logging.NOTSET)` that recreates *structlog*'s defaults on top of standard library's `logging`. It optionally also configures `logging` to log to standard out at the passed log level. #428

- `structlog.processors.EventRenamer` allows you to rename the hitherto hard-coded event dict key `event` to something else. Optionally, you can rename another key to `event` at the same time, too. So adding `EventRenamer(to="msg", replace_by="_event")` to your processor pipeline will rename the standard `event` key to `msg` and then rename the `_event` key to `event`. This allows you to use the `event` key in your own log files and to have consistent log message keys across languages.

- `structlog.dev.ConsoleRenderer(event_key="event")` now allows to customize the name of the key that is used for the log message.

Changed

- `structlog.make_filtering_bound_logger()` now returns a method with the same signature for all log levels, whether they are active or not. This ensures that invalid calls to inactive log levels are caught immediately and don't explode once the log level changes. #401

- `structlog.PrintLogger` – that is used by default – now uses `print()` for printing, making it a better citizen for interactive terminal applications. #399

- `structlog.testing.capture_logs` now works for already initialized bound loggers. #408

- `structlog.processors.format_exc_info()` is no longer a function, but an instance of `structlog.processors.ExceptionRenderer`. Its behavior has not changed. #407

- The default configuration now includes the `structlog.contextvars.merge_contextvars` processor. That means you can use `structlog.contextvars` features without configuring *structlog*.

Fixed

- Overloaded the `bind`, `unbind`, `try_unbind` and `new` methods in the `FilteringBoundLogger` Protocol. This makes it easier to use objects of type `FilteringBoundLogger` in a typed context. #392

- Monkeypatched `sys.stdout`s are now handled more gracefully by `ConsoleRenderer` (that's used by default). #404

- `structlog.stdlib.render_to_log_kwargs()` now correctly handles the presence of `exc_info`, `stack_info`, and `stackLevel` in the event dictionary. They are transformed into proper keyword arguments instead of putting them into the `extra` dictionary. #424, #427

9.3.4 21.5.0 - 2021-12-16

Added

- Added the `structlog.processors.LogfmtRenderer` processor to render log lines using the *logfmt* format. #376

- Added the `structlog.stdlib.ExtraAdder` processor that adds extra attributes of `logging.LogRecord` objects to the event dictionary. This processor can be used for adding data passed in the `extra` parameter of the `logging` module's log methods to the event dictionary. #209, #377

- Added the `structlog.processor.CallsiteParameterAdder` processor that adds parameters of the callsite that an event dictionary originated from to the event dictionary. This processor can be used to enrich events dictionaries with information such as the function name, line number and filename that an event dictionary originated from. #380

9.3.5 21.4.0 - 2021-11-25

Added

- Added the `structlog.threadlocal.bound_threadlocal` and `structlog.contextvars.bound_contextvars` decorator/context managers to temporarily bind key-value pairs to a thread-local and context-local context. #371

Fixed

- Fixed import when running in optimized mode (`PYTHONOPTIMIZE=2` or `python -OO`). #373

9.3.6 21.3.0 - 2021-11-20

Added

- `structlog.dev.ConsoleRenderer` now has `sort_keys` boolean parameter that allows to disable the sorting of keys on output. #358

Changed

- *structlog* switched its packaging to *flit*. Users shouldn't notice a difference, but (re-)packagers might.

- `structlog.stdlib.AsyncBoundLogger` now determines the running loop when logging, not on instantiation. That has a minor performance impact, but makes it more robust when loops change (e.g. `aiohttp.web.run_app()`), or you want to use `sync_bl` *before* a loop has started.

Fixed

- `structlog.processors.TimeStamper` now works well with *FreezeGun* even when it gets applied before the loggers are configured. #364

- `structlog.stdlib.ProcessorFormatter` now has a *processors* argument that allows to define a processor chain to run over *all* log entries.

  Before running the chain, two additional keys are added to the event dictionary: `_record` and `_from_structlog`. With them it's possible to extract information from `logging.LogRecord`s and differentiate between *structlog* and `logging` log entries while processing them.

  The old *processor* (singular) parameter is now deprecated, but no plans exist to remove it. #365

9.3.7 21.2.0 - 2021-10-12

Added

- `structlog.threadlocal.get_threadlocal()` and `structlog.contextvars.get_contextvars()` can now be used to get a copy of the current thread-local/context-local context that has been bound using `structlog.threadlocal.bind_threadlocal()` and `structlog.contextvars.bind_contextvars()`. #331, #337

- `structlog.threadlocal.get_merged_threadlocal(bl)` and `structlog.contextvars.get_merged_contextvars(bl)` do the same, but also merge the context from a bound logger *bl*. Same pull requests as previous change.

- `structlog.contextvars.bind_contextvars()` now returns a mapping of keys to `contextvars.Token`s, allowing you to reset values using the new `structlog.contextvars.reset_contextvars()`. #339

- Exception rendering in `structlog.dev.ConsoleLogger` is now configurable using the `exception_formatter` setting. If either the *Rich* or the *better-exceptions* package is present, *structlog* will use them for pretty-printing tracebacks. *Rich* takes precedence over *better-exceptions* if both are present.

  This only works if `format_exc_info` is **absent** in the processor chain. #330, #349

- The final processor can now return a `bytearray` (additionally to `str` and `bytes`). #344

Changed

- To implement pretty exceptions (see Changes below), `structlog.dev.ConsoleRenderer` now formats exceptions itself.

  Make sure to remove `format_exc_info` from your processor chain if you configure *structlog* manually. This change is not really breaking, because the old use-case will keep working as before. However if you pass `pretty_exceptions=True` (which is the default if either `rich` or `better-exceptions` is installed), a warning will be raised and the exception will be rendered without prettification.

- All use of *Colorama* on non-Windows systems has been excised. Thus, colors are now enabled by default in `structlog.dev.ConsoleRenderer` on non-Windows systems. You can keep using *Colorama* to customize colors, of course. #345

Fixed

- *structlog* is now importable if `sys.stdout` is `None` (e.g. when running using `pythonw`). #313

9.3.8 21.1.0 - 2021-02-18

Changed

- `structlog.dev.ConsoleRenderer` will now look for a `logger_name` key if no `logger` key is set. #295

Fixed

- `structlog.threadlocal.wrap_dict()` now has a correct type annotation. #290
- Fix isolation in `structlog.contextvars`. #302
- The default configuration and loggers are pickleable again. #301

9.3.9 20.2.0 - 2020-12-31

Removed

- Python 2.7 and 3.5 aren't supported anymore. The package meta data should ensure that you keep getting 20.1.0 on those versions. #244

Deprecated

- Accessing the `_context` attribute of a bound logger is now deprecated. Please use the new `structlog.get_context()`.

Added

- *structlog* has now type hints for all of its APIs! Since *structlog* is highly dynamic and configurable, this led to a few concessions like a specialized `structlog.stdlib.get_logger()` whose only difference to `structlog.get_logger()` is that it has the correct type hints.

  We consider them provisional for the time being – i.e. the backwards-compatibility does not apply to them in its full strength until we feel we got it right. Please feel free to provide feedback! #223, #282

- Added `structlog.make_filtering_logger` that can be used like `configure(wrapper_class=make_filtering_bound_lo INFO))`. It creates a highly optimized bound logger whose inactive methods only consist of a `return None`. This is now also the default logger.

- As a complement, `structlog.stdlib.add_log_level()` can now additionally be imported as `structlog.processors.add_log_level` since it just adds the method name to the event dict.

- Added `structlog.BytesLogger` to avoid unnecessary encoding round trips. Concretely this is useful with *orjson* which returns bytes. #271

- The final processor now also may return bytes that are passed untouched to the wrapped logger.

- `structlog.get_context()` allows you to retrieve the original context of a bound logger. #266,

- Added `structlog.testing.CapturingLogger` for more unit testing goodness.

- Added `structlog.stdlib.AsyncBoundLogger` that executes logging calls in a thread executor and therefore doesn't block. #245

Changed

- The default bound logger (`wrapper_class`) if you don't configure *structlog* has changed. It's mostly compatible with the old one but a few uncommon methods like `log`, `failure`, or `err` don't exist anymore.

  You can regain the old behavior by using `structlog.configure(wrapper_class=structlog.BoundLogger)`.

  Please note that due to the various interactions between settings, it's possible that you encounter even more errors. We **strongly** urge you to always configure all possible settings since the default configuration is *not* covered by our backwards-compatibility policy.

- `structlog.processors.add_log_level()` is now part of the default configuration.

- `structlog.stdlib.ProcessorFormatter` no longer uses exceptions for control flow, allowing `foreign_pre_chain` processors to use `sys.exc_info()` to access the real exception.

Fixed

- `structlog.PrintLogger` now supports `copy.deepcopy()`. #268

9.3.10 20.1.0 - 2020-01-28

Deprecated

- This is the last version to support Python 2.7 (including PyPy) and 3.5. All following versions will only support Python 3.6 or later.

Added

- Added a new module `structlog.contextvars` that allows to have a global but context-local *structlog* context the same way as with `structlog.threadlocal` since 19.2.0. #201, #236

- Added a new module `structlog.testing` for first class testing support. The first entry is the context manager `capture_logs()` that allows to make assertions about structured log calls. #14, #234

- Added `structlog.threadlocal.unbind_threadlocal()`. #239

Fixed

- The logger created by `structlog.get_logger()` is not detected as an abstract method anymore, when attached to an abstract base class. #229

- *Colorama* isn't initialized lazily on Windows anymore because it breaks rendering. #232, #242

9.3.11 19.2.0 - 2019-10-16

Removed

- Python 3.4 is not supported anymore. It has been unsupported by the Python core team for a while now and its PyPI downloads are negligible.

  It's very unlikely that *structlog* will break under 3.4 anytime soon, but we don't test it anymore.

Added

- Full Python 3.8 support for `structlog.stdlib`.

- Added more pass-through properties to `structlog.stdlib.BoundLogger`. To makes it easier to use it as a drop-in replacement for `logging.Logger`. #198

- Added new processor `structlog.dev.set_exc_info()` that will set `exc_info=True` if the method's name is `exception` and `exc_info` isn't set at all. *This is only necessary when the standard library integration is not used*. It fixes the problem that in the default configuration, `structlog.get_logger().exception("hi")` in an `except` block would not print the exception without passing `exc_info=True` to it explicitly. #130, #173, #200, #204

- Added a new thread-local API that allows binding values to a thread-local context explicitly without affecting the default behavior of `bind()`. #222, #225

- Added *pass_foreign_args* argument to `structlog.stdlib.ProcessorFormatter`. It allows to pass a foreign log record's *args* attribute to the event dictionary under the `positional_args` key. #228

Changed

- `structlog.stdlib.ProcessorFormatter` now takes a logger object as an optional keyword argument. This makes `ProcessorFormatter` work properly with `stuctlog.stdlib.filter_by_level()`. #219

- `structlog.dev.ConsoleRenderer` now calls `str()` on the event value. #221

Fixed

- `structlog.dev.ConsoleRenderer` now uses no colors by default, if *Colorama* is not available. #215

- `structlog.dev.ConsoleRenderer` now initializes *Colorama* lazily, to prevent accidental side-effects just by importing *structlog*. #210

- A best effort has been made to make as much of *structlog* pickleable as possible to make it friendlier with `multiprocessing` and similar libraries. Some classes can only be pickled on Python 3 or using the dill library though and that is very unlikely to change.

  So far, the configuration proxy, `structlog.processor.TimeStamper`, `structlog.BoundLogger`, `structlog.PrintLogger` and `structlog.dev.ConsoleRenderer` have been made pickleable. Please report if you need any another class fixed. #126

9.3.12 19.1.0 - 2019-02-02

Added

- `structlog.ReturnLogger` and `structlog.PrintLogger` now have a `fatal()` log method. #181

Changed

- As announced in 18.1.0, `pip install -e .[dev]` now installs all development dependencies. Sorry for the inconveniences this undoubtedly will cause!
- *structlog* now tolerates passing through `dicts` to stdlib logging. #187, #188, #189

Fixed

- Under certain (rather unclear) circumstances, the frame extraction could throw an `SystemError:` `error return without exception set`. A workaround has been added. #174

9.3.13 18.2.0 - 2018-09-05

Added

- Added `structlog.stdlib.add_log_level_number()` processor that adds the level *number* to the event dictionary. Can be used to simplify log filtering. #151
- `structlog.processors.JSONRenderer` now allows for overwriting the *default* argument of its serializer. #77, #163
- Added `try_unbind()` that works like `unbind()` but doesn't raise a `KeyError` if one of the keys is missing. #171

9.3.14 18.1.0 - 2018-01-27

Deprecated

- 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 development (`pip` will report an error).

Added

- `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
- Added `structlog.is_configured()` to check whether or not *structlog* has been configured.

- Added `structlog.get_config()` to introspect current configuration.

Changed

- Empty strings are valid events now. #110

- `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 `dict`s on Python 3.6+ and PyPy since they are ordered by default.

Fixed

- Do not encapsulate Twisted failures twice with newer versions of Twisted. #144

9.3.15 17.2.0 - 2017-05-15

Added

- `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

Fixed

- `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

9.3.16 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.

Added

- 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.LogRecord`s. #79, #105

- 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 (human-friendly behavior) in `ConsoleRenderer`. #94

- Added *colors* argument to `structlog.dev.ConsoleRenderer` and made it the default renderer. #78

Changed

- 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 backwards-compatibility policy that explicitly excludes default settings.

- UNIX epoch timestamps from `structlog.processors.TimeStamper` are more precise now.

- Positional arguments are now removed even if they are empty. #82

9.3.17 Fixed

- Fixed bug with Python 3 and `structlog.stdlib.BoundLogger.log()`. Error log level was not reproducible and was logged as exception one time out of two. #92

9.3.18 16.1.0 - 2016-05-24

Removed

- 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.

Added

- Added 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

Fixed

- 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

9.3.19 16.0.0 - 2016-01-28

Added

- Added `structlog.dev.ConsoleRenderer` that renders the event dictionary aligned and with colors.

- Added `structlog.processors.UnicodeDecoder` that will decode all byte string values in an event dictionary to Unicode.

- Added `serializer` parameter to `structlog.processors.JSONRenderer` which allows for using different (possibly faster) JSON encoders than the standard library.

Changed

- `structlog.processors.ExceptionPrettyPrinter` and `structlog.processors.format_exc_info` now support passing of Exceptions on Python 3.

- *six* is now used for compatibility.

Fixed

- The context is now cleaned up when exiting `structlog.threadlocal.tmp_bind` in case of exceptions. #64

- Be more more lenient about missing `__name__`s. #62

9.3.20 15.3.0 - 2015-09-25

Added

- Officially support Python 3.5.

- Added `structlog.ReturnLogger.failure` and `structlog.PrintLogger.failure` as preparation for the new Twisted logging system.

Fixed

- Tolerate frames without a `__name__`, better. #58

9.3.21 15.2.0 - 2015-06-10

Added

- Added option to specify target key in `structlog.processors.TimeStamper` processor. #51

Changed

- 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.

- Better support of `logging.Logger.exception` within *structlog*. #52

Fixed

- Prevent Twisted's `log.err` from quoting strings rendered by `structlog.twisted.JSONRenderer`.

9.3.22 15.1.0 - 2015-02-24

Fixed

- Tolerate frames without a `__name__` when guessing callsite names.

9.3.23 15.0.0 - 2015-01-23

Added

- Added `structlog.stdlib.add_log_level` and `structlog.stdlib.add_logger_name` processors. #44
- Added `structlog.stdlib.BoundLogger.log`. #42
- Added `structlog.stdlib.BoundLogger.exception`. #22

Changed

- 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

9.3.24 0.4.2 - 2014-07-26

Removed

- 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.

Added

- Officially support Python 3.4.
- Allow final processor to return a dictionary. See the adapting chapter. #26
- Test Twisted-related code on Python 3 (with some caveats).

Fixed

- 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.
- `from structlog import *` works now (but you still shouldn't use it).

9.3.25 0.4.1 - 2013-12-19

Changed

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

Fixed

- Various doc fixes.

9.3.26 0.4.0 - 2013-11-10

Added

- Added `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

9.3.27 0.3.2 - 2013-09-27

Fixed

- Fix stdlib's name guessing.

9.3.28 0.3.1 - 2013-09-26

Fixed

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

9.3.29 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`.

9.3.30 0.2.0 - 2013-09-17

Added

- Add `key_order` option to `structlog.processors.KeyValueRenderer` for more predictable log entries with any `dict` class.

- Enhance Twisted support by offering JSONification of non-structlog log entries.

- Allow for custom serialization in `structlog.twisted.JSONRenderer` without abusing `__repr__`.

Changed

- Promote to stable, thus henceforth a strict backwards-compatibility policy is put into effect.

- `structlog.PrintLogger` now uses proper I/O routines and is thus viable not only for examples but also for production.

9.3.31 0.1.0 - 2013-09-16

Initial release.

TEN

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

S

Symbols

A

B

C

D

E