django-structlog

Release 5.1.0

Apr 23, 2023

Contents

django-structlog is a structured logging integration for Django project using structlog
Logging will then produce additional cohesive metadata on each logs that makes it easier to track events or incidents.

CHAPTER 1

Additional Popular Integrations

1.1 Django REST framework

`Django REST framework` is supported by default. But when using it with `rest_framework.authentication.TokenAuthentication` (or other DRF authentications) `user_id` will be only be in `request_finished` and `request_failed` instead of each logs.
See #37 for details.

1.2 Celery

Celery's task logging requires additional configurations, see documentation for details.

CHAPTER 2

Logging comparison

2.1 Standard logging:

```
>>> import logging
>>> logger = logging.get_logger(__name__)
>>> logger.info("An error occurred")
```

```
An error occurred
```

Well… ok

2.2 With django-structlog and flat_line:

```
>>> import structlog
>>> logger = structlog.get_logger(__name__)
>>> logger.info("an_error_occurred", bar="Buz")
```

```
timestamp='2019-04-13T19:39:31.089925Z' level='info' event='an_error_occurred' logger=
↪'my_awesome_project.my_awesome_module' request_id='3a8f801c-072b-4805-8f38-
↪e1337f363ed4' user_id=1 ip='0.0.0.0' bar='Buz'
```

Then you can search with commands like:

```
$ cat logs/flat_line.log | grep request_id='3a8f801c-072b-4805-8f38-e1337f363ed4'
```

2.3 With django-structlog and json

```
>>> import structlog
>>> logger = structlog.get_logger(__name__)
>>> logger.info("an_error_occurred", bar="Buz")
```

```
{"request_id": "3a8f801c-072b-4805-8f38-e1337f363ed4", "user_id": 1, "ip": "0.0.0.0",
↪"event": "an_error_occurred", "timestamp": "2019-04-13T19:39:31.089925Z", "logger":
↪"my_awesome_project.my_awesome_module", "level": "info", "bar": "Buz"}
```

(continues on next page)

(continued from previous page)

```
```

Then you can search with commands like:

```
$ cat logs/json.log | jq '.[] | select(.request_id="3a8f801c-072b-4805-8f38-
↪e1337f363ed4")' -s
```

CHAPTER 3

Contents, indices and tables

3.1 Getting Started

These steps will show how to integrate the middleware to your awesome application.

3.1.1 Installation

Install the library

```
pip install django-structlog
```

Add middleware

```
MIDDLEWARE = [
    # ...
    'django_structlog.middlewares.RequestMiddleware',
]
```

Add appropriate structlog configuration to your `settings.py`

```
import structlog

LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "json_formatter": {
            "()": structlog.stdlib.ProcessorFormatter,
            "processor": structlog.processors.JSONRenderer(),
        },
        "plain_console": {
            "()": structlog.stdlib.ProcessorFormatter,
            "processor": structlog.dev.ConsoleRenderer(),
        },
        "key_value": {
            "()": structlog.stdlib.ProcessorFormatter,
            "processor": structlog.processors.KeyValueRenderer(key_order=['timestamp',
↪ 'level', 'event', 'logger']),
```

(continues on next page)

(continued from previous page)

```
            },
        },
        "handlers": {
            "console": {
                "class": "logging.StreamHandler",
                "formatter": "plain_console",
            },
            "json_file": {
                "class": "logging.handlers.WatchedFileHandler",
                "filename": "logs/json.log",
                "formatter": "json_formatter",
            },
            "flat_line_file": {
                "class": "logging.handlers.WatchedFileHandler",
                "filename": "logs/flat_line.log",
                "formatter": "key_value",
            },
        },
        "loggers": {
            "django_structlog": {
                "handlers": ["console", "flat_line_file", "json_file"],
                "level": "INFO",
            },
            # Make sure to replace the following logger's name for yours
            "django_structlog_demo_project": {
                "handlers": ["console", "flat_line_file", "json_file"],
                "level": "INFO",
            },
        }
}

structlog.configure(
    processors=[
        structlog.contextvars.merge_contextvars,
        structlog.stdlib.filter_by_level,
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    logger_factory=structlog.stdlib.LoggerFactory(),
    cache_logger_on_first_use=True,
)
```

Start logging with `structlog` instead of `logging`.

```
import structlog
logger = structlog.get_logger(__name__)
```

3.1.2 Extending Request Log Metadata

By default only a `request_id` and the `user_id` are bound from the request but pertinent log metadata may vary from a project to another.

If you need to add more metadata from the request you can implement a convenient signal receiver to bind them. You can also override existing bound metadata the same way.

```
from django.dispatch import receiver

from django_structlog.signals import bind_extra_request_metadata
import structlog


@receiver(bind_extra_request_metadata)
def bind_user_email(request, logger, **kwargs):
    structlog.contextvars.bind_contextvars(user_email=getattr(request.user, 'email', '
↪'))
```

3.1.3 Standard Loggers

It is also possible to log using standard python logger.
In your formatters, add the `foreign_pre_chain` section, and then add `structlog.contextvars.merge_contextvars`:

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "json_formatter": {
            "()": structlog.stdlib.ProcessorFormatter,
            "processor": structlog.processors.JSONRenderer(),
            # Add this section:
            "foreign_pre_chain": [
                structlog.contextvars.merge_contextvars, # <---- add this
                # customize the rest as you need
                structlog.processors.TimeStamper(fmt="iso"),
                structlog.stdlib.add_logger_name,
                structlog.stdlib.add_log_level,
                structlog.stdlib.PositionalArgumentsFormatter(),
            ],
        },
    },
    ...
 }
```

3.2 Celery Integration

3.2.1 Getting Started with Celery

In order to be able to support celery you need to configure both your webapp and your workers

Replace your requirements

First of all, make sure your `django-structlog` installation knows you use `celery` in order to validate compatibility with your installed version. See Installing "Extras" for more information.
Replace `django-structlog` with `django-structlog[celery]` in your `requirements.txt`.

```
django-structlog[celery]==X.Y.Z
```

Add CeleryMiddleWare in your web app

In your settings.py

```
MIDDLEWARE = [
    # ...
    'django_structlog.middlewares.RequestMiddleware',
    'django_structlog.middlewares.CeleryMiddleware',
]
```

Initialize Celery Worker with DjangoStructLogInitStep

In your celery AppConfig's module.

```
import logging

import structlog
from celery import Celery
from celery.signals import setup_logging
from django_structlog.celery.steps import DjangoStructLogInitStep

app = Celery("your_celery_project")

# A step to initialize django-structlog
app.steps['worker'].add(DjangoStructLogInitStep)
```

Configure celery's logger

In the same file as before

```
@setup_logging.connect
def receiver_setup_logging(loglevel, logfile, format, colorize, **kwargs):  # pragma:␣
↪no cover
    logging.config.dictConfig(
        {
           "version": 1,
           "disable_existing_loggers": False,
           "formatters": {
               "json_formatter": {
                   "()": structlog.stdlib.ProcessorFormatter,
                   "processor": structlog.processors.JSONRenderer(),
               },
               "plain_console": {
                   "()": structlog.stdlib.ProcessorFormatter,
                   "processor": structlog.dev.ConsoleRenderer(),
               },
               "key_value": {
                   "()": structlog.stdlib.ProcessorFormatter,
                   "processor": structlog.processors.KeyValueRenderer(key_order=[
↪'timestamp', 'level', 'event', 'logger']),
               },
           },
           "handlers": {
               "console": {
                   "class": "logging.StreamHandler",
                   "formatter": "plain_console",
               },
               "json_file": {
```

(continues on next page)

(continued from previous page)

```
                    "class": "logging.handlers.WatchedFileHandler",
                    "filename": "logs/json.log",
                    "formatter": "json_formatter",
                },
                "flat_line_file": {
                    "class": "logging.handlers.WatchedFileHandler",
                    "filename": "logs/flat_line.log",
                    "formatter": "key_value",
                },
            },
            "loggers": {
                "django_structlog": {
                    "handlers": ["console", "flat_line_file", "json_file"],
                    "level": "INFO",
                },
                "django_structlog_demo_project": {
                    "handlers": ["console", "flat_line_file", "json_file"],
                    "level": "INFO",
                },
            }
        }
    )

    structlog.configure(
        processors=[
            structlog.contextvars.merge_contextvars,
            structlog.stdlib.filter_by_level,
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.stdlib.PositionalArgumentsFormatter(),
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.UnicodeDecoder(),
            structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
        ],
        logger_factory=structlog.stdlib.LoggerFactory(),
        cache_logger_on_first_use=True,
    )
```

3.2.2 Signals

modify_context_before_task_publish

You can connect to `modify_context_before_task_publish` signal in order to modify the metadata before it is stored in the task's message.
By example you can strip down the `context` to keep only some of the keys:

```
@receiver(signals.modify_context_before_task_publish)
def receiver_modify_context_before_task_publish(sender, signal, context, **kwargs):
    keys_to_keep = {"request_id", "parent_task_id"}
    new_dict = {key_to_keep: context[key_to_keep] for key_to_keep in keys_to_keep if␣
↪key_to_keep in context}
    context.clear()
    context.update(new_dict)
```

bind_extra_task_metadata

You can optionally connect to `bind_extra_task_metadata` signal in order to bind more metadata to the logger or override existing bound metadata. This is called in celery's `receiver_task_pre_run`.

```
from django_structlog.celery import signals
import structlog

@receiver(signals.bind_extra_task_metadata)
def receiver_bind_extra_request_metadata(sender, signal, task=None, logger=None,␣
↪**kwargs):
    structlog.contextvars.bind_contextvars(correlation_id=task.request.correlation_id)
```

3.3 API documentation

3.3.1 django_structlog

`django-structlog` is a structured logging integration for `Django` project using `structlog`.

class `django_structlog.middlewares.`**`RequestMiddleware`**`(`*get_response*`)`

Bases: `django_structlog.middlewares.request.BaseRequestMiddleWare`

`RequestMiddleware` adds request metadata to `structlog`'s logger context automatically.

```
>>> MIDDLEWARE = [
...     # ...
...     'django_structlog.middlewares.RequestMiddleware',
... ]
```

`async_capable = True`

`sync_capable = True`

`django_structlog.signals.`**`bind_extra_request_metadata = <django.dispatch.dispatcher.Signal`**

Signal to add extra `structlog` bindings from `django`'s request.

Parameters **`logger`** – the logger to bind more metadata or override existing bound metadata

```
>>> from django.dispatch import receiver
>>> from django_structlog import signals
>>> import structlog
>>>
>>> @receiver(signals.bind_extra_request_metadata)
... def bind_user_email(request, logger, **kwargs):
...     structlog.contextvars.bind_contextvars(user_email=getattr(request.user,
↪'email', ''))
```

`django_structlog.signals.`**`bind_extra_request_finished_metadata = <django.dispatch.dispatche`**

Signal to add extra `structlog` bindings from `django`'s finished request and response.

Parameters

- **`logger`** – the logger to bind more metadata or override existing bound metadata
- **`response`** – the response resulting of the request

```
>>> from django.dispatch import receiver
>>> from django_structlog import signals
>>> import structlog
>>>
>>> @receiver(signals.bind_extra_request_finished_metadata)
... def bind_user_email(request, logger, response, **kwargs):
...     structlog.contextvars.bind_contextvars(user_email=getattr(request.user,
↪'email', ''))
```

django_structlog.signals.**bind_extra_request_failed_metadata = <django.dispatch.dispatcher.S**

Signal to add extra `structlog` bindings from `django`'s failed request and exception.

Parameters

- **`logger`** – the logger to bind more metadata or override existing bound metadata
- **`exception`** – the exception resulting of the request

```
>>> from django.dispatch import receiver
>>> from django_structlog import signals
>>> import structlog
>>>
>>> @receiver(signals.bind_extra_request_failed_metadata)
... def bind_user_email(request, logger, exception, **kwargs):
...     structlog.contextvars.bind_contextvars(user_email=getattr(request.user,
↪'email', ''))
```

django_structlog.signals.**update_failure_response = <django.dispatch.dispatcher.Signal obje**

Signal to update response failure response before it is returned.

Parameters

- **`request`** – the request returned by the view
- **`response`** – the response resulting of the request
- **`logger`** – the logger
- **`exception`** – the exception

```
>>> from django.dispatch import receiver
>>> from django_structlog import signals
>>> import structlog
>>>
>>> @receiver(signals.update_failure_response)
... def add_request_id_to_error_response(request, response, logger, exception,␣
↪**kwargs):
...     context = structlog.contextvars.get_merged_contextvars(logger)
...     response['X-Request-ID'] = context["request_id"]
```

3.3.2 django_structlog.celery

`celery` integration for `django_structlog`.

class django_structlog.celery.middlewares.**CeleryMiddleware**(*get_response=None*)

Bases: `object`

async_capable = True

`CeleryMiddleware` initializes `celery` signals to pass `django`'s request information to `celery` worker's logger.

```
>>> MIDDLEWARE = [
...     # ...
...     'django_structlog.middlewares.RequestMiddleware',
...     'django_structlog.middlewares.CeleryMiddleware',
... ]
```

sync_capable = True

django_structlog.celery.signals.**bind_extra_task_metadata = <django.dispatch.dispatcher.Sign**

Signal to add extra `structlog` bindings from `celery`'s task.

Parameters

- **`task`** – the celery task being run
- **`logger`** – the logger to bind more metadata or override existing bound metadata

```
>>> from django.dispatch import receiver
>>> from django_structlog.celery import signals
```

(continues on next page)

(continued from previous page)

```
>>> import structlog
>>>
>>> @receiver(signals.bind_extra_task_metadata)
... def receiver_bind_extra_request_metadata(sender, signal, task=None,␣
↪logger=None, **kwargs):
...     structlog.contextvars.bind_contextvars(correlation_id=task.request.
↪correlation_id)
```

django_structlog.celery.signals.**modify_context_before_task_publish = <django.dispatch.dispa**

Signal to modify context passed over to `celery` task's context. You must modify the `context` dict.

Parameters **context** – the context dict that will be passed over to the task runner's logger

```
>>> from django.dispatch import receiver
>>> from django_structlog.celery import signals
>>>
>>> @receiver(signals.modify_context_before_task_publish)
... def receiver_modify_context_before_task_publish(sender, signal, context,␣
↪**kwargs):
...     keys_to_keep = {"request_id", "parent_task_id"}
...     new_dict = {
...         key_to_keep: context[key_to_keep]
...         for key_to_keep in keys_to_keep
...         if key_to_keep in context
...     }
...     context.clear()
...     context.update(new_dict)
```

django_structlog.celery.signals.**pre_task_succeeded = <django.dispatch.dispatcher.Signal ob**

Signal to add `structlog` bindings from `celery`'s successful task.

Parameters

- **logger** – the logger to bind more metadata or override existing bound metadata
- **result** – result of the succeeding task

```
>>> from django.dispatch import receiver
>>> from django_structlog.celery import signals
>>> import structlog
>>>
>>> @receiver(signals.pre_task_succeeded)
... def receiver_pre_task_succeeded(sender, signal, logger=None, result=None,␣
↪**kwargs):
...     structlog.contextvars.bind_contextvars(result=str(result))
```

3.4 Events and Metadata

3.4.1 Django's RequestMiddleware

Request Events

Event	Type	Description
request_started	INFO	Django received a request
request_finished	INFO	request completed normally
request_failed	ERROR	unhandled exception occurred

Request Bound Metadata

These metadata are repeated on each log of the current request and will be also be repeated in all children Celery tasks.

Key	Value
request_id	UUID for the request or value of `X-Request-ID` HTTP header when provided
correlation_id	value of `X-Correlation-ID` HTTP header when provided
user_id	user's id or None (requires django.contrib.auth.middleware.AuthenticationMiddleware) DRF: it will only be in `request_finished` and `request_failed` events **If you need to override the bound `user_id`, it has to be done in all thr** • *`django_structlog.signals.bind_extra_request_metadata`* • *`django_structlog.signals.bind_extra_request_finished_metadata`* • *`django_structlog.signals.bind_extra_request_failed_metadata`*
ip	request's ip

To bind more metadata or override existing metadata from request see *Extending Request Log Metadata*

Request Events Metadata

These metadata appear once along with their associated event

Event	Key	Value
request_started	request	request as string
request_started	user_agent	request's user agent
request_finished	code	request's status code
request_failed	exception	exception traceback (requires format_exc_info)

3.4.2 CeleryMiddleware

Task Events

Event	Type	Description
task_enqueued	INFO	A task was enqueued by request or another task
task_retrying	WARNING	Worker retry task
task_succeeded	INFO	Task completed successfully
task_failed	ERROR/INFO*	Task failed
task_revoked	WARNING	Task was canceled
task_not_found	ERROR	Celery app did not discover the requested task
task_task_rejected	ERROR	Task could not be enqueued

* if task threw an expected exception, it will logged as `INFO`. See Celery's Task.throws

Task Bound Metadata

These metadata are repeated on each log of the current task and will be also be repeated in all children Celery tasks. Take note that all the caller's logger bound metadata are also bound to the task's logger.

Key	Value
task_id	UUID of the current task
parent_task_id	UUID of the parent's task (if any)

To bind more metadata or override existing metadata from task see *Signals*

Task Event Metadata

These metadata appear once along with their associated event

Event	Key	Value
task_enqueued	child_task_id	id of the task being enqueued
task_enqueued	child_task_name	name of the task being enqueued
task_retrying	reason	reason for retry
task_failed	error	exception as string
task_failed	exception*	exception's traceback
task_revoked	terminated	Set to True if the task was terminated
task_revoked	signum	see Celery's documentation
task_revoked	expired	see Celery's documentation

* if task threw an expected exception, `exception` will be omitted. See Celery's Task.throws

3.5 Example outputs

3.5.1 Flat lines file (`logs/flat_lines.log`)

```
timestamp='2019-04-13T19:39:29.321453Z' level='info' event='request_started' logger=
↪'django_structlog.middlewares.request' request_id='c53dff1d-3fc5-4257-a78a-
↪9a567c937561' user_id=1 ip='0.0.0.0' request=GET / user_agent='Mozilla/5.0␣
↪(Macintosh; Intel Mac OS X 10_14_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
↪73.0.3683.86 Safari/537.36'
timestamp='2019-04-13T19:39:29.345207Z' level='info' event='request_finished' logger=
↪'django_structlog.middlewares.request' request_id='c53dff1d-3fc5-4257-a78a-
↪9a567c937561' user_id=1 ip='0.0.0.0' code=200
timestamp='2019-04-13T19:39:31.086155Z' level='info' event='request_started' logger=
↪'django_structlog.middlewares.request' request_id='3a8f801c-072b-4805-8f38-
↪e1337f363ed4' user_id=1 ip='0.0.0.0' request=POST /success_task user_agent='Mozilla/
↪5.0 (Macintosh; Intel Mac OS X 10_14_4) AppleWebKit/537.36 (KHTML, like Gecko)␣
↪Chrome/73.0.3683.86 Safari/537.36'
timestamp='2019-04-13T19:39:31.089925Z' level='info' event='Enqueuing successful task
↪' logger='django_structlog_demo_project.home.views' request_id='3a8f801c-072b-4805-
↪8f38-e1337f363ed4' user_id=1 ip='0.0.0.0'
timestamp='2019-04-13T19:39:31.147590Z' level='info' event='task_enqueued' logger=
↪'django_structlog.middlewares.celery' request_id='3a8f801c-072b-4805-8f38-
↪e1337f363ed4' user_id=1 ip='0.0.0.0' child_task_id='6b11fd80-3cdf-4de5-acc2-
↪3fd4633aa654'
timestamp='2019-04-13T19:39:31.153081Z' level='info' event='This is a successful task
↪' logger='django_structlog_demo_project.taskapp.celery' task_id='6b11fd80-3cdf-4de5-
↪acc2-3fd4633aa654' request_id='3a8f801c-072b-4805-8f38-e1337f363ed4' user_id=1 ip=
↪'0.0.0.0'
timestamp='2019-04-13T19:39:31.160043Z' level='info' event='request_finished' logger=
↪'django_structlog.middlewares.request' request_id='3a8f801c-072b-4805-8f38-
↪e1337f363ed4' user_id=1 ip='0.0.0.0' code=201
```

(continues on next page)

(continued from previous page)

```
timestamp='2019-04-13T19:39:31.162372Z' level='info' event='task_succeed' logger=
↪'django_structlog.middlewares.celery' task_id='6b11fd80-3cdf-4de5-acc2-3fd4633aa654
↪' request_id='3a8f801c-072b-4805-8f38-e1337f363ed4' user_id=1 ip='0.0.0.0' result=
↪'None'
```

3.5.2 Json file (logs/json.log)

```
{"request_id": "c53dff1d-3fc5-4257-a78a-9a567c937561", "user_id": 1, "ip": "0.0.0.0",
↪"request": "GET /", "user_agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_4)␣
↪AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.36", "event":
↪"request_started", "timestamp": "2019-04-13T19:39:29.321453Z", "logger": "django_
↪structlog.middlewares.request", "level": "info"}
{"request_id": "c53dff1d-3fc5-4257-a78a-9a567c937561", "user_id": 1, "ip": "0.0.0.0",
↪"code": 200, "event": "request_finished", "timestamp": "2019-04-13T19:39:29.345207Z
↪", "logger": "django_structlog.middlewares.request", "level": "info"}
{"request_id": "3a8f801c-072b-4805-8f38-e1337f363ed4", "user_id": 1, "ip": "0.0.0.0",
↪"request": "POST /success_task", "user_agent": "Mozilla/5.0 (Macintosh; Intel Mac␣
↪OS X 10_14_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.
↪36", "event": "request_started", "timestamp": "2019-04-13T19:39:31.086155Z", "logger
↪": "django_structlog.middlewares.request", "level": "info"}
{"request_id": "3a8f801c-072b-4805-8f38-e1337f363ed4", "user_id": 1, "ip": "0.0.0.0",
↪"event": "Enqueuing successful task", "timestamp": "2019-04-13T19:39:31.089925Z",
↪"logger": "django_structlog_demo_project.home.views", "level": "info"}
{"request_id": "3a8f801c-072b-4805-8f38-e1337f363ed4", "user_id": 1, "ip": "0.0.0.0",
↪"child_task_id": "6b11fd80-3cdf-4de5-acc2-3fd4633aa654", "event": "task_enqueued",
↪"timestamp": "2019-04-13T19:39:31.147590Z", "logger": "django_structlog.middlewares.
↪celery", "level": "info"}
{"task_id": "6b11fd80-3cdf-4de5-acc2-3fd4633aa654", "request_id": "3a8f801c-072b-4805-
↪8f38-e1337f363ed4", "user_id": 1, "ip": "0.0.0.0", "event": "This is a successful␣
↪task", "timestamp": "2019-04-13T19:39:31.153081Z", "logger": "django_structlog_demo_
↪project.taskapp.celery", "level": "info"}
{"request_id": "3a8f801c-072b-4805-8f38-e1337f363ed4", "user_id": 1, "ip": "0.0.0.0",
↪"code": 201, "event": "request_finished", "timestamp": "2019-04-13T19:39:31.160043Z
↪", "logger": "django_structlog.middlewares.request", "level": "info"}
{"task_id": "6b11fd80-3cdf-4de5-acc2-3fd4633aa654", "request_id": "3a8f801c-072b-4805-
↪8f38-e1337f363ed4", "user_id": 1, "ip": "0.0.0.0", "result": "None", "event": "task_
↪succeed", "timestamp": "2019-04-13T19:39:31.162372Z", "logger": "django_structlog.
↪middlewares.celery", "level": "info"}
```

3.6 Running the tests

Note: For the moment redis is needed to run the tests. The easiest way is to start docker demo's redis.

```
docker compose up -d redis
pip install -r requirements.txt
env CELERY_BROKER_URL=redis://0.0.0.0:6379 DJANGO_SETTINGS_MODULE=config.settings.
↪test pytest test_app
env CELERY_BROKER_URL=redis://0.0.0.0:6379 DJANGO_SETTINGS_MODULE=config.settings.
↪test_demo_app pytest django_structlog_demo_project
docker compose stop redis
```

3.7 Development

3.7.1 Prerequisites

- docker

3.7.2 Installation

```
$ git clone https://github.com/jrobichaud/django-structlog.git
$ cd django-structlog
$ pip install -r requirements.txt
$ pre-commit install
```

3.7.3 Start Demo App

```
$ docker compose up --build
```

- WSGI server: Open `http://127.0.0.1:8000/` in your browser.
- ASGI server: Open `http://127.0.0.1:8001/` in your browser.

3.7.4 Building, Serving and Testing the Documentation Locally

```
$ docker compose -p django-structlog-docs -f docker-compose.docs.yml up --build
Serving on http://127.0.0.1:5000
```

3.8 Demo app

```
docker compose up --build
```

Open `http://127.0.0.1:8000/` in your browser.
Navigate while looking into the log files and shell's output.

3.9 Change Log

3.9.1 5.1.0 (April 22, 2023)

New:

- Add new signal `django_structlog.signals.update_failure_response` allowing to modify the response in case of failure. See #231. Special thanks to @HMaker.

3.9.2 5.0.2 (April 16, 2023)

See: *Upgrading to 5.0+*
Fixes:

- Fix regression in 5.0.0 and 5.0.1 where exceptions were not logged as `error` but as `info`. See #226. Special thanks to @ntap-fge.

Rollbacks from 5.0.0:

- Rollback removal of `django_structlog.signals.bind_extra_request_failed_metadata`. Relates the above fix.

3.9.3 5.0.1 (March 24, 2023)

See: *Upgrading to 5.0+*

Changes:

- minimum requirements change for `asgiref` to 3.6.0. See #209. Special thanks to @adinsoon.

3.9.4 5.0.0 (March 23, 2023)

See: *Upgrading to 5.0+*

Changes:

- `RequestMiddleware` and `CeleryMiddleware` now properly support async views

Removed:

- *(Rolled back in 5.0.2)* `django_structlog.signals.bind_extra_request_failed_metadata`

Deprecates:

- `django_structlog.middlewares.request_middleware_router`
- `django_structlog.middlewares.requests.AsyncRequestMiddleware`
- `django_structlog.middlewares.requests.SyncRequestMiddleware`

3.9.5 4.1.1 (February 7, 2023)

New:

- Add `django_structlog.middlewares.request_middleware_router` to choose automatically between Async or Sync middleware

Rollbacks from 4.1.0:

- Rollback `RequestMiddleware` not being a class anymore, its an internal `SyncRequestMiddleware`

Others:

- Migrate project to `pyproject.toml` instead of `setup.py`
- Add *asgi* server to demo project see *Development*.

3.9.6 4.1.0 (February 4, 2023)

New:

- Add async view support. See #180. Special thanks to @DamianMel.

Changes:

- `RequestMiddleware` is no longer a class but a function due to async view support. This should only affect projects using the middleware not as intended. If this cause you problems, please refer to this issue #183, the documentation or feel free to open a new issue. Special thanks to @gvangool for pointing that out.

Others:

- Add colours in log in the demo project. See 63bdb4d to update your projects. Special thanks to @RoscoeTheDog.
- Upgrade or remove various development packages

3.9.7 4.0.1 (October 25, 2022)

New:

- Add support to `python` 3.11. See #142. Special thanks to @jairhenrique.

3.9.8 4.0.0 (October 22, 2022)

See: *Upgrading to 4.0+*

Changes:

- `django-structlog` will now on follow LTS versions of Python, Django, and Celery. See #110. Special thanks to @jairhenrique for his convincing arguments.

New:

- You can now install `django-structlog` with `celery` extra. Specifying `django-structlog[celery]==4.0.0` in `requirements.txt` will make sure your `celery`'s version is compatible.

Others:

- Upgrade or remove various development packages
- Upgrade local development environment from python 3.7 to 3.10 and from django 3.2 to django 4.1
- Added a gh-pages

3.9.9 3.0.1 (August 2, 2022)

Fixes:

- `AttributeError` with custom User without `pk`. See #80. Special thanks to @mlegner.

Others:

- Add `dependabot` to manage dependencies. See #83. Special thanks to @jairhenrique.
- Upgrade various development packages

3.9.10 3.0.0 (August 1, 2022)

See: *Upgrading to 3.0+*

Changes:

- **`django-structlog` now uses `structlog.contextvars` instead of `structlog.threadlocal`. See the upgra**
  - removed `django_structlog.processors.inject_context_dict`
  - minimum requirements change to `python` 3.7+
  - minimum requirements change to `structlog` 21.5

New:

- Add python 3.10, celery 5.2 and django 4.0 to the test matrix.

Others:

- Remove `wrapper_class` from the configuration

3.9.11 2.2.0 (November 18, 2021)

Changes:

- Requests were logged as `<WSGIRequest: GET '/'>` (as an object) and now they are logged like this `GET /` (as a string). See #72. Special thanks to @humitos.

3.9.12 2.1.3 (September 28, 2021)

Fixes:

- Implement Celery Task.throws' behaviour of logging expected exception as `INFO` with no tracebacks. See #62 and #70. Special thanks to @meunomemauricio.

3.9.13 2.1.2 (August 31, 2021)

Fixes:

- `django.core.exceptions.PermissionDenied` is no longer logged as 500 but 403. See #68. Special thanks to @rabbit-aaron.

3.9.14 2.1.1 (June 22, 2021)

Others:

- Add `django` 3.2 and `python` 3.9 to the test matrix and `pypi` metadata. See #65. Special thanks to @kashewnuts.

3.9.15 2.1.0 (November 26, 2020)

New:

- `django_structlog.processors.inject_context_dict` for standard python loggers. See #24. Special thanks to @debfx.

3.9.16 2.0.0 (November 25, 2020)

Upgrade:

- There are necessary configuration changes needed. See *Upgrading to 2.0+* for the details.

Changes:

- No longer add `error` and `error_traceback`. See #55 and *Upgrading to 2.0+*. Special thanks to @debfx.

Fixes:

- Fix crash when request's user is `None` for django-oauth-toolkit. See #56. Special thanks to @nicholasamorim.

3.9.17 1.6.3 (November 11, 2020)

Improvements:

- Call stack of exception in log is now an appropriate string. See #54. Special thanks to @debfx.

3.9.18 1.6.2 (October 4, 2020)

Fixes:

- Fix UUID as User pk causing issues. See #52 #45 and #51. Special thanks to @fadedDexofan.

3.9.19 1.6.1 (August 13, 2020)

Fixes:

- Removed `providing_args` from signals to fix django 4.0 deprecation warnings introduced by django 3.1. See #44. Special thanks to @ticosax.
- Fix `sender` of `signals.pre_task_succeeded`
- Documented signal parameters in doc strings and `API documentation` to replace `providing_args`

Others:

- Add `django` 3.0 and 3.1 to the test matrix and `pypi` supported frameworks metadata
- Fix reference of the previous ci in the documentation

3.9.20 1.6.0 (June 17, 2020)

Changes:

- `task_succeed` is now `task_succeeded`. Special thanks to @PawelMorawian.
- Remove `result` from `task_succeeded` log (may be added back, see below). Special thanks to @PawelMorawian as well.
- Add `django_structlog.celery.signals.pre_task_succeeded`. To be able to bind `result` if someone really needs it.

3.9.21 1.5.5 (June 16, 2020)

New:

- Add `bind_extra_request_finished_metadata` and `bind_extra_request_failed_metadata`. See #39. Special thanks to @prik2693.

3.9.22 1.5.4 (June 15, 2020)

Improvements:

- Remove redundant `DJANGO_STRUCTLOG_LOG_USER_IN_REQUEST_FINISHED` setting and just always make sure `user_id` is in `request_finished` and `request_failed` instead. See #37.

3.9.23 1.5.3 (June 15, 2020)

New:

- Add `DJANGO_STRUCTLOG_LOG_USER_IN_REQUEST_FINISHED` setting to support Django REST framework. See #37. Special thanks to @immortaleeb.

3.9.24 1.5.2 (April 2, 2020)

New:

- Add `modify_context_before_task_publish` signal.

3.9.25 1.5.1 (March 18, 2020)

Improvements:

- Allow to override celery task metadata from binding. See #32 and #33. Special thanks to @chiragjn

3.9.26 1.5.0 (March 6, 2020)

Improvements:

- Add support for celery 3. See #26 and #31. Special thanks to @chiragjn and @prik2693

3.9.27 1.4.1 (February 8, 2020)

New:

- Bind `X-Correlation-ID` HTTP header's value as `correlation_id` when provided in request.

3.9.28 1.4.0 (February 7, 2020)

New:

- Use `X-Request-ID` HTTP header's value as `request_id` when provided in request. See #22. Special thanks to @jairhenrique

3.9.29 1.3.5 (December 23, 2019)

New:

- Add python 3.8, celery 4.4 and django 3.0 to the test matrix.

Improvements:

- Extract `test_app` from `django_structlog_demo_app` in order to test `django_structlog` all by itself
- Improve CI execution speed by merging stages
- Upgrade a few development depencencies

3.9.30 1.3.4 (November 27, 2019)

Bugfix:

- Exception logging not working properly with `DEBUG = False`. See #19. Special thanks to @danpalmer

3.9.31 1.3.3 (October 6, 2019)

Bugfix:

- Fix support of different primary key for `User` model. See #13. Special thanks to @dhararon

3.9.32 1.3.2 (September 21, 2019)

Improvements:

- Add support of projects without `AuthenticationMiddleware`. See #9. Special thanks to @dhararon

3.9.33 1.3.1 (September 4, 2019)

Bugfixes:

- Remove extraneous `rest-framework` dependency introduced by #7. See #8 . Special thanks to @ghickman

3.9.34 1.3.0 (September 3, 2019)

Improvements:

- Improve django uncaught exception formatting. See #7. Special thanks to @paulstuartparker

3.9.35 1.2.3 (May 18, 2019)

Bugfixes:

- Fix `structlog` dependency not being installed

Improvements:

- Use black code formatter

3.9.36 1.2.2 (May 13, 2019)

Improvements:

- Use appropriate packaging

3.9.37 1.2.1 (May 8, 2019)

Bugfixes:

- Fix missing license file to be included in distribution

3.9.38 1.2.0 (May 8, 2019)

Changes:

- In the event `task_enqueued`, `task_id` and `task_name` are renamed `child_task_id` and `child_task_name` respectively to avoid override of `task_id` in nested tasks.

3.9.39 1.1.6 (May 8, 2019)

New:

- Add `task_name` when a task is enqueued

3.9.40 1.1.5 (May 8, 2019)

New:

- Add support of tasks calling other tasks (introducing `parent_task_id`)

Bugfixes:

- Fix missing packages

3.9.41 1.1.4 (April 22, 2019)

Improvements:

- Wheel distribution

3.9.42 1.1.3 (April 22, 2019)

Improvements:

- api documentation
- code documentation

3.9.43 1.1.2 (April 19, 2019)

Changes:

- Rewrite the log texts as events

3.9.44 1.1.1 (April 18, 2019)

New:

- Add `celery` signal `signals.bind_extra_task_metadata`

3.9.45 1.1 (April 16, 2019)

New:

- Add `celery` tasks support

3.9.46 1.0.4 to 1.0.7 (April 14, 2019)

New:

- Automated releases with tags on `travis`

3.9.47 1.0.3 (April 14, 2019)

Bugfixes:

- Add `bind_extra_request_metadata` documentation

3.9.48 1.0.2 (April 13, 2019)

Bugfixes:

- Tweaked documentation.

3.9.49 1.0.0 (April 13, 2019)

New:

- Fist public release.

3.10 Upgrade Guide

3.10.1 Upgrading to 5.0+

Minimum requirements

- requires asgiref 3.6+

Change you may need to do

Make sure you use `django_structlog.middlewares.RequestMiddleware`

If you used any of the experimental async or sync middlewares, you do not need to anymore. Make sure you use `django_structlog.middlewares.RequestMiddleware` instead of any of the other request middlewares commented below:

```
MIDDLEWARE += [
    # "django_structlog.middlewares.request_middleware_router", # <- remove
    # "django_structlog.middlewares.requests.SyncRequestMiddleware", # <- remove
    # "django_structlog.middlewares.requests.AsyncRequestMiddleware", # <- remove
    "django_structlog.middlewares.RequestMiddleware", # <- make sure you use this one
    "django_structlog.middlewares.CeleryMiddleware",
]
```

They will be removed in another major version.

3.10.2 Upgrading to 4.0+

`django-structlog` drops support of django below 3.2.

Minimum requirements

- requires django 3.2+
- requires python 3.7+
- requires structlog 21.4.0+
- (optionally) requires celery 5.1+

Changes if you use `celery`

You can now install `django-structlog` explicitly with `celery` extra in order to validate the compatibility with your version of `celery`.

```
django-structlog[celery]==4.0.0
```

See Installing "Extras" for more information about this `pip` feature.

3.10.3 Upgrading to 3.0+

`django-structlog` now use structlog.contextvars.bind_contextvars instead of `threadlocal`.

Minimum requirements

- requires python 3.7+
- requires structlog 21.4.0+

Changes you need to do

1. Update structlog settings

- add `structlog.contextvars.merge_contextvars` as first `processors`
- remove `context_class=structlog.threadlocal.wrap_dict(dict),`
- (if you use standard loggers) add `structlog.contextvars.merge_contextvars` in *foreign_pre_chain*
- (if you use standard loggers) remove `django_structlog.processors.inject_context_dict,`

```
structlog.configure(
    processors=[
        structlog.contextvars.merge_contextvars, # <---- add this
        structlog.stdlib.filter_by_level,
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    # context_class=structlog.threadlocal.wrap_dict(dict), # <---- remove this
    logger_factory=structlog.stdlib.LoggerFactory(),
    cache_logger_on_first_use=True,
)

# If you use standard logging
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "json_formatter": {
            "()": structlog.stdlib.ProcessorFormatter,
            "processor": structlog.processors.JSONRenderer(),
            "foreign_pre_chain": [
                structlog.contextvars.merge_contextvars, # <---- add this
                # django_structlog.processors.inject_context_dict, # <---- remove this
                structlog.processors.TimeStamper(fmt="iso"),
                structlog.stdlib.add_logger_name,
                structlog.stdlib.add_log_level,
                structlog.stdlib.PositionalArgumentsFormatter(),
            ],
        },
    },
    ...
 }
```

2. Replace all `logger.bind` with `structlog.contextvars.bind_contextvars`

```
@receiver(bind_extra_request_metadata)
def bind_user_email(request, logger, **kwargs):
    # logger.bind(user_email=getattr(request.user, 'email', ''))
    structlog.contextvars.bind_contextvars(user_email=getattr(request.user, 'email', '
↪'))
```

3.10.4 Upgrading to 2.0+

`django-structlog` was originally developed using the debug configuration ExceptionPrettyPrinter which led to incorrect handling of exception.

- remove `structlog.processors.ExceptionPrettyPrinter(),` of your processors.
- make sure you have `structlog.processors.format_exc_info,` in your processors if you want appropriate exception logging.

3.11 Authors

- **Jules Robichaud-Gagnon** - *Initial work* - jrobichaud

See also the list of contributors who participated in this project.

3.12 Acknowledgments

- Big thanks to @ferd for his bad opinions that inspired the author enough to spend time on this library.
- This issue helped the author to figure out how to integrate `structlog` in Django.
- This stack overflow question was also helpful.

3.13 Licence

- genindex
- modindex
- search

Python Module Index

Index