Raven Documentation

Release 5.5.0

David Cramer

July 22, 2015

Contents

For pairing Sentry up with Python you can use the Raven for Python (raven-python) library. It is the official standalone Python client for Sentry. It can be used with any modern Python interpreter be it CPython 2.x or 3.x, PyPy or Jython. It's an Open Source project and available under a very liberal BSD license.

CHAPTER 1

Installation

If you haven't already, start by downloading Raven. The easiest way is with *pip*:

```
pip install raven --upgrade
```

CHAPTER 2

Configuring the Client

Settings are specified as part of the initialization of the client. The client is a class that can be instanciated with a specific configuration and all reporting can then happen from the instance of that object. Typically an instance is created somewhere globally and then imported as necessary. For getting started all you need is your DSN:

```
from raven import Client
client = Client('___DSN___')
```

CHAPTER 3

Capture an Error

The most basic use for raven is to record one specific error that occurs:

```
from raven import Client

client = Client('___DSN___')

try:
    1 / 0
except ZeroDivisionError:
    client.captureException()
```

CHAPTER 4

Adding Context

The raven client internally keeps a thread local mapping that can carry additional information. Whenever a message is submitted to Sentry that additional data will be passed along. This context is available as *client.context* and can be modified or cleared.

Example usage:

```
def handle_request(request):
    client.context.merge({'user': {
        'email': request.user.email
    }})
    try:
        ...
    finally:
        client.context.clear()
```

CHAPTER 5

Deep Dive

Raven Python is more than that however. To dive deeper into what it does, how it works and how it integrates into other systems there is more to discover:

5.1 Basic Usage

This gives a basic overview of how to use the raven client with Python directly.

5.1.1 Capture an Error

The most basic use for raven is to record one specific error that occurs:

```
from raven import Client

client = Client('___DSN___')

try:
    1 / 0
except ZeroDivisionError:
    client.captureException()
```

5.1.2 Reporting an Event

To report an arbitrary event you can use the *capture()* method. This is the most low-level method available. In most cases you would want to use the *captureMessage()* method instead however which directly reports a message:

```
client.captureMessage('Something went fundamentally wrong')
```

5.1.3 Adding Context

The raven client internally keeps a thread local mapping that can carry additional information. Whenever a message is submitted to Sentry that additional data will be passed along.

For instance if you use a web framework, you can use this to inject additional information into the context. The basic primitive for this is the *context* attribute. It provides a *merge()* and *clear()* function that can be used:

```
def handle_request(request):
    client.context.merge({'user': {
        'email': request.user.email
    }})
    try:
        ...
    finally:
        client.context.clear()
```

5.1.4 Testing the Client

Once you've got your server configured, you can test the Raven client by using its CLI:

```
raven test ___DSN___
```

If you've configured your environment to have `SENTRY_DSN` available, you can simply drop the optional DSN argument:

```
raven test
```

You should get something like the following, assuming you're configured everything correctly:

```
$ raven test sync+___DSN___
Using DSN configuration:
  sync+___DSN___

Client configuration:
  servers         : ___API_URL___/api/store/
  project         : ___PROJECT_ID___
  public_key      : ___PUBLIC_KEY___
  secret_key      : ___SECRET_KEY___

Sending a test message... success!
```

5.2 Advanced Usage

This covers some advanced usage scenarios for raven Python.

5.2.1 Alternative Installations

If you want to use the latest git version you can get it from the github repository:

```
git clone https://github.com/getsentry/raven-python
pip install raven-python
```

Certain additional features can be installed by defining the feature when `pip` installing it. For instance to install all dependencies needed to use the Flask integration, you can depend on `raven[flask]`:

```
pip install raven[flask]
```

For more information refer to the individual integration documentation.

5.2.2 Configuring the Client

Settings are specified as part of the initialization of the client. The client is a class that can be instantiated with a specific configuration and all reporting can then happen from the instance of that object. Typically an instance is created somewhere globally and then imported as necessary.

```
from raven import Client

# Read configuration from the ``SENTRY_DSN`` environment variable
client = Client()

# Manually specify a DSN
client = Client('___DSN___')
```

A reasonably configured client should generally include a few additional settings:

```
import raven

client = raven.Client(
    dsn='___DSN___'

    # inform the client which parts of code are yours
    # include_paths=['my.app']
    include_paths=[__name__.split('.', 1)[0]],

    # pass along the version of your application
    # release='1.0.0'
    # release=raven.fetch_package_version('my-app')
    release=raven.fetch_git_sha(os.path.dirname(__file__)),
)
```

New in version 5.2.0: The *fetch_package_version* and *fetch_git_sha* helpers.

5.2.3 The Sentry DSN

The Python client supports one additional modification to the regular DSN values which is the choice of the transport. To select a specific transport, the DSN needs to be prepended with the name of the transport. For instance to select the `gevent` transport, the following DSN would be used:

```
'gevent+___DSN___'
```

For more information see Transports.

5.2.4 Client Arguments

The following are valid arguments which may be passed to the Raven client:

dsn
A Sentry compatible DSN as mentioned before:

```
dsn = '___DSN___'
```

site
An optional, arbitrary string to identify this client installation:

```
site = 'my site name'
```

name

This will override the `server_name` value for this installation. Defaults to `socket.gethostname()`:

```
name = 'sentry_rocks_' + socket.gethostname()
```

release

The version of your application. This will map up into a Release in Sentry:

```
release = '1.0.3'
```

exclude_paths

Extending this allow you to ignore module prefixes when we attempt to discover which function an error comes from (typically a view):

```
exclude_paths = [
    'django',
    'sentry',
    'raven',
    'lxml.objectify',
]
```

include_paths

For example, in Django this defaults to your list of `INSTALLED_APPS`, and is used for drilling down where an exception is located:

```
include_paths = [
    'django',
    'sentry',
    'raven',
    'lxml.objectify',
]
```

max_list_length

The maximum number of items a list-like container should store.

If an iterable is longer than the specified length, the left-most elements up to length will be kept.

Note: This affects sets as well, which are unordered.

```
list_max_length = 50
```

string_max_length

The maximum characters of a string that should be stored.

If a string is longer than the given length, it will be truncated down to the specified size:

```
string_max_length = 200
```

auto_log_stacks

Should Raven automatically log frame stacks (including locals) for all calls as it would for exceptions:

```
auto_log_stacks = True
```

processors

A list of processors to apply to events before sending them to the Sentry server. Useful for sending additional global state data or sanitizing data that you want to keep off of the server:

```
processors = (
    'raven.processors.SanitizePasswordsProcessor',
)
```

5.2.5 Sanitizing Data

Several processors are included with Raven to assist in data sanitiziation. These are configured with the `processors` value.

5.2.6 A Note on uWSGI

If you're using uWSGI you will need to add `enable-threads` to the default invocation, or you will need to switch off of the threaded default transport.

5.3 Integrations

The Raven Python module also comes with integration for some commonly used libraries to automatically capture errors from common environments. This means that once you have such an integration configured you typically do not need to report errors manually.

Some integrations allow specifying these in a standard configuration, otherwise they are generally passed upon instantiation of the Sentry client.

5.3.1 Bottle

Bottle is a microframework for Python. Raven supports this framework through the WSGI integration.

Setup

The first thing you'll need to do is to disable catchall in your Bottle app:

```
import bottle

app = bottle.app()
app.catchall = False
```

Note: Bottle will not propagate exceptions to the underlying WSGI middleware by default. Setting catchall to False disables that.

Sentry will then act as Middleware:

```
from raven import Client
from raven.contrib.bottle import Sentry
client = Client('___DSN___')
app = Sentry(app, client)
```

Usage

Once you've configured the Sentry application you need only call run with it:

```
run(app=app)
```

If you want to send additional events, a couple of shortcuts are provided on the Bottle request app object.

Capture an arbitrary exception by calling `captureException`:

```
try:
    1 / 0
except ZeroDivisionError:
    request.app.sentry.captureException()
```

Log a generic message with `captureMessage`:

```
request.app.sentry.captureMessage('Hello, world!')
```

5.3.2 Celery

Celery is a distributed task queue system for Python built on AMQP principles. For Celery built-in support by Raven is provided but it requires some manual configuraiton.

To capture errors, you need to register a couple of signals to hijack Celery error handling:

```
from raven import Client
from raven.contrib.celery import register_signal, register_logger_signal

client = Client('___DSN___')

# register a custom filter to filter out duplicate logs
register_logger_signal(client)

# hook into the Celery error handler
register_signal(client)

# The register_logger_signal function can also take an optional argument
# `loglevel` which is the level used for the handler created.
# Defaults to `logging.ERROR`
register_logger_signal(client, loglevel=logging.INFO)
```

A more complex version to encapsulate behavior:

```
import celery
import raven
from raven.contrib.celery import register_signal, register_logger_signal

class Celery(celery.Celery):

    def on_configure(self):
        client = raven.Client('___DSN___')

        # register a custom filter to filter out duplicate logs
        register_logger_signal(client)

        # hook into the Celery error handler
        register_signal(client)

app = Celery(__name__)
app.config_from_object('django.conf:settings')
```

5.3.3 Django

Django is arguably Python's most popular web framework. Support is built into Raven but needs some configuration. While older versions of Django will likely work, officially only version 1.4 and newer are supported.

Setup

Using the Django integration is as simple as adding `raven.contrib.django.raven_compat` to your installed apps:

```
INSTALLED_APPS = (
    'raven.contrib.django.raven_compat',
)
```

Note: This causes Raven to install a hook in Django that will automatically report uncaught exceptions.

Additional settings for the client are configured using the `RAVEN_CONFIG` dictionary:

```
import raven

RAVEN_CONFIG = {
    'dsn': '___DSN___',
    # If you are using git, you can also automatically configure the
    # release based on the git info.
    'release': raven.fetch_git_sha(os.path.dirname(__file__)),
}
```

Once you've configured the client, you can test it using the standard Django management interface:

```
python manage.py raven test
```

You'll be referencing the client slightly differently in Django as well:

```
from raven.contrib.django.raven_compat.models import client

client.captureException()
```

Using with Raven.js

A Django template tag is provided to render a proper public DSN inside your templates, you must first load `raven`:

```
{% load raven %}
```

Inside your template, you can now use:

```
<script>Raven.config('{% sentry_public_dsn %}').install()</script>
```

By default, the DSN is generated in a protocol relative fashion, e.g. `//public@example.com/1`. If you need a specific protocol, you can override:

```
{% sentry_public_dsn 'https' %}
```

Integration with `logging`

To integrate with the standard library's `logging` module the following config can be used:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'root': {
        'level': 'WARNING',
        'handlers': ['sentry'],
```

```
    },
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s '
                      '%(process)d %(thread)d %(message)s'
        },
    },
    'handlers': {
        'sentry': {
            'level': 'ERROR',
            'class': 'raven.contrib.django.raven_compat.handlers.SentryHandler',
        },
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'verbose'
        }
    },
    'loggers': {
        'django.db.backends': {
            'level': 'ERROR',
            'handlers': ['console'],
            'propagate': False,
        },
        'raven': {
            'level': 'DEBUG',
            'handlers': ['console'],
            'propagate': False,
        },
        'sentry.errors': {
            'level': 'DEBUG',
            'handlers': ['console'],
            'propagate': False,
        },
    },
}
```

Usage

Logging usage works the same way as it does outside of Django, with the addition of an optional `request` key in the extra data:

```
logger.error('There was some crazy error', exc_info=True, extra={
    # Optionally pass a request and we'll grab any information we can
    'request': request,
})
```

404 Logging

In certain conditions you may wish to log 404 events to the Sentry server. To do this, you simply need to enable a Django middleware:

```
MIDDLEWARE_CLASSES = (
  'raven.contrib.django.raven_compat.middleware.Sentry404CatchMiddleware',
  ...,
) + MIDDLEWARE_CLASSES
```

It is recommended to put the middleware at the top, so that any only 404s that bubbled all the way up get logged. Certain middlewares (e.g. flatpages) capture 404s and replace the response.

Message References

Sentry supports sending a message ID to your clients so that they can be tracked easily by your development team. There are two ways to access this information, the first is via the `X-Sentry-ID` HTTP response header. Adding this is as simple as appending a middleware to your stack:

```
MIDDLEWARE_CLASSES = MIDDLEWARE_CLASSES + (
  # We recommend putting this as high in the chain as possible
  'raven.contrib.django.raven_compat.middleware.SentryResponseErrorIdMiddleware',
  ...,
)
```

Another alternative method is rendering it within a template. By default, Sentry will attach `request.sentry` when it catches a Django exception. In our example, we will use this information to modify the default `500.html` which is rendered, and show the user a case reference ID. The first step in doing this is creating a custom `handler500()` in your `urls.py` file:

```
from django.conf.urls.defaults import *

from django.views.defaults import page_not_found, server_error
from django.template import Context, loader
from django.http import HttpResponseServerError

def handler500(request):
    """500 error handler which includes ``request`` in the context.

    Templates: `500.html`
    Context: None
    """

    t = loader.get_template('500.html') # You need to create a 500.html template.
    return HttpResponseServerError(t.render(Context({
        'request': request,
    })))
```

Once we've successfully added the `request` context variable, adding the Sentry reference ID to our `500.html` is simple:

```
<p>You've encountered an error, oh noes!</p>
{% if request.sentry.id %}
    <p>If you need assistance, you may reference this error as
    <strong>{{ request.sentry.id }}</strong>.</p>
{% endif %}
```

WSGI Middleware

If you are using a WSGI interface to serve your app, you can also apply a middleware which will ensure that you catch errors even at the fundamental level of your Django application:

```
from raven.contrib.django.raven_compat.middleware.wsgi import Sentry
from django.core.handlers.wsgi import WSGIHandler

application = Sentry(WSGIHandler())
```

Additional Settings

SENTRY_CLIENT

In some situations you may wish for a slightly different behavior to how Sentry communicates with your server. For this, Raven allows you to specify a custom client:

```
SENTRY_CLIENT = 'raven.contrib.django.raven_compat.DjangoClient'
```

SENTRY_CELERY_LOGLEVEL

If you are also using Celery, there is a handler being automatically registered for you that captures the errors from workers. The default logging level for that handler is `logging.ERROR` and can be customized using this setting:

```
SENTRY_CELERY_LOGLEVEL = logging.INFO
RAVEN_CONFIG = {
    'CELERY_LOGLEVEL': logging.INFO
}
```

Caveats

The following things you should keep in mind when using Raven with Django.

Error Handling Middleware

If you already have middleware in place that handles `process_exception()` you will need to take extra care when using Sentry.

For example, the following middleware would suppress Sentry logging due to it returning a response:

```
class MyMiddleware(object):
    def process_exception(self, request, exception):
        return HttpResponse('foo')
```

To work around this, you can either disable your error handling middleware, or add something like the following:

```
from django.core.signals import got_request_exception

class MyMiddleware(object):
    def process_exception(self, request, exception):
        # Make sure the exception signal is fired for Sentry
        got_request_exception.send(sender=self, request=request)
        return HttpResponse('foo')
```

Note that this technique may break unit tests using the Django test client (`django.test.client.Client`) if a view under test generates a `Http404` or `PermissionDenied` exception, because the exceptions won't be translated into the expected 404 or 403 response codes.

Or, alternatively, you can just enable Sentry responses:

```
from raven.contrib.django.raven_compat.models import sentry_exception_handler

class MyMiddleware(object):
    def process_exception(self, request, exception):
        # Make sure the exception signal is fired for Sentry
        sentry_exception_handler(request=request)
        return HttpResponse('foo')
```

Gunicorn

If you are running Django with gunicorn and using the `gunicorn` executable, instead of the `run_gunicorn` management command, you will need to add a hook to gunicorn to activate Raven:

```
from django.core.management import call_command

def when_ready(server):
    call_command('validate')
```

Circus

If you are running Django with circus and chaussette you will also need to add a hook to circus to activate Raven:

```
from django.conf import settings
from django.core.management import call_command

def run_raven(*args, **kwargs):
    """Set up raven for django by running a django command.
    It is necessary because chaussette doesn't run a django command.
    """
    if not settings.configured:
        settings.configure()

    call_command('validate')
    return True
```

And in your circus configuration:

```
[socket:dwebapp]
host = 127.0.0.1
port = 8080

[watcher:dwebworker]
cmd = chaussette --fd $(circus.sockets.dwebapp) dproject.wsgi.application
use_sockets = True
numprocesses = 2
hooks.after_start = dproject.hooks.run_raven
```

5.3.4 Flask

Flask is a popular Python micro webframework. Support for Flask is provided by Raven directly but for some dependencies you need to install raven with the flask feature set.

Installation

If you haven't already, install raven with its explicit Flask dependencies:

```
pip install raven[flask]
```

Setup

The first thing you'll need to do is to initialize Raven under your application:

```
from raven.contrib.flask import Sentry
sentry = Sentry(app, dsn='___DSN___')
```

If you don't specify the `dsn` value, we will attempt to read it from your environment under the `SENTRY_DSN` key.

Extended Setup

You can optionally configure logging too:

```
import logging
from raven.contrib.flask import Sentry
sentry = Sentry(app, logging=True, level=logging.ERROR)
```

Building applications on the fly? You can use Raven's `init_app` hook:

```
sentry = Sentry(dsn='http://public_key:secret_key@example.com/1')

def create_app():
    app = Flask(__name__)
    sentry.init_app(app)
    return app
```

You can pass parameters in the `init_app` hook:

```
sentry = Sentry()

def create_app():
    app = Flask(__name__)
    sentry.init_app(app, dsn='http://public_key:secret_key@example.com/1',
                    logging=True, level=logging.ERROR)
    return app
```

Settings

Additional settings for the client can be configured using `SENTRY_<setting name>` in your application's configuration:

```
class MyConfig(object):
    SENTRY_DSN = '___DSN___'
    SENTRY_INCLUDE_PATHS = ['myproject']
```

If Flask-Login is used by your application (including Flask-Security), user information will be captured when an exception or message is captured. By default, only the `id` (current_user.get_id()), `is_authenticated`, and `is_anonymous` is captured for the user. If you would like additional attributes on the `current_user` to be captured, you can configure them using `SENTRY_USER_ATTRS`:

```
class MyConfig(object):
    SENTRY_USER_ATTRS = ['username', 'first_name', 'last_name', 'email']
```

`email` will be captured as `sentry.interfaces.User.email`, and any additionl attributes will be available under `sentry.interfaces.User.data`

You can specify the types of exceptions that should not be reported by Sentry client in your application by setting the `RAVEN_IGNORE_EXCEPTIONS` configuration value on your Flask app configuration:

```
class MyExceptionType(Exception):
    def __init__(self, message):
        super(MyExceptionType, self).__init__(message)

app = Flask(__name__)
app.config["RAVEN_IGNORE_EXCEPTIONS"] = [MyExceptionType]
```

Usage

Once you've configured the Sentry application it will automatically capture uncaught exceptions within Flask. If you want to send additional events, a couple of shortcuts are provided on the Sentry Flask middleware object.

Capture an arbitrary exception by calling `captureException`:

```
try:
    1 / 0
except ZeroDivisionError:
    sentry.captureException()
```

Log a generic message with `captureMessage`:

```
sentry.captureMessage('hello, world!')
```

Getting the last event id

If possible, the last Sentry event ID is stored in the request context `g.sentry_event_id` variable. This allow to present the user an error ID if have done a custom error 500 page.

```
<h2>Error 500</h2>
{% if g.sentry_event_id %}
<p>The error identifier is {{ g.sentry_event_id }}</p>
{% endif %}
```

Dealing with proxies

When your Flask application is behind a proxy such as nginx, Sentry will use the remote address from the proxy, rather than from the actual requesting computer. By using `ProxyFix` from werkzeug.contrib.fixers the Flask `.wsgi_app` can be modified to send the actual `REMOTE_ADDR` along to Sentry.

```
from werkzeug.contrib.fixers import ProxyFix
app.wsgi_app = ProxyFix(app.wsgi_app)
```

This may also require changes to the proxy configuration to pass the right headers if it isn't doing so already.

5.3.5 Logbook

Raven provides a logbook handler which will pipe messages to Sentry.

First you'll need to configure a handler:

```
from raven.handlers.logbook import SentryHandler

# Manually specify a client
client = Client(...)
handler = SentryHandler(client)
```

You can also automatically configure the default client with a DSN:

```
# Configure the default client
handler = SentryHandler('___DSN___')
```

Finally, bind your handler to your context:

```
from raven.handlers.logbook import SentryHandler

client = Client(...)
sentry_handler = SentryHandler(client)
with sentry_handler.applicationbound():
    # everything logged here will go to sentry.
    ...
```

5.3.6 Logging

Sentry supports the ability to directly tie into the `logging` module. To use it simply add `SentryHandler` to your logger.

First you'll need to configure a handler:

```
from raven.handlers.logging import SentryHandler

# Manually specify a client
client = Client(...)
handler = SentryHandler(client)
```

You can also automatically configure the default client with a DSN:

```
# Configure the default client
handler = SentryHandler('___DSN___')
```

Finally, call the `setup_logging()` helper function:

```
from raven.conf import setup_logging

setup_logging(handler)
```

Another option is to use `logging.config.dictConfig`:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,

    'formatters': {
        'console': {
            'format': '[%(asctime)s][%(levelname)s] %(name)s '
                      '%(filename)s:%(funcName)s:%(lineno)d | %(message)s',
            'datefmt': '%H:%M:%S',
            },
        },

    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'console'
            },
```

```
        'sentry': {
            'level': 'ERROR',
            'class': 'raven.handlers.logging.SentryHandler',
            'dsn': '___DSN___',
            },
        },

    'loggers': {
        '': {
            'handlers': ['console', 'sentry'],
            'level': 'DEBUG',
            'propagate': False,
            },
        'your_app': {
            'level': 'DEBUG',
            'propagate': True,
        },
    }
}
```

Usage

A recommended pattern in logging is to simply reference the modules name for each logger, so for example, you might at the top of your module define the following:

```
import logging
logger = logging.getLogger(__name__)
```

You can also use the `exc_info` and `extra={'stack': True}` arguments on your `log` methods. This will store the appropriate information and allow Sentry to render it based on that information:

```
# If you're actually catching an exception, use `exc_info=True`
logger.error('There was an error, with a stacktrace!', exc_info=True)

# If you don't have an exception, but still want to capture a
# stacktrace, use the `stack` arg
logger.error('There was an error, with a stacktrace!', extra={
    'stack': True,
})
```

Note: Depending on the version of Python you're using, `extra` might not be an acceptable keyword argument for a logger's `.exception()` method (`.debug()`, `.info()`, `.warning()`, `.error()` and `.critical()` should work fine regardless of Python version). This should be fixed as of Python 3.2. Official issue here: http://bugs.python.org/issue15541.

While we don't recommend this, you can also enable implicit stack capturing for all messages:

```
client = Client(..., auto_log_stacks=True)
handler = SentryHandler(client)

logger.error('There was an error, with a stacktrace!')
```

You may also pass additional information to be stored as meta information with the event. As long as the key name is not reserved and not private (_foo) it will be displayed on the Sentry dashboard. To do this, pass it as `data` within your `extra` clause:

```
logger.error('There was some crazy error', exc_info=True, extra={
    # Optionally you can pass additional arguments to specify request info
    'culprit': 'my.view.name',

    'data': {
        # You may specify any values here and Sentry will log and output them
        'username': request.user.username,
    }
})
```

Note: The `url` and `view` keys are used internally by Sentry within the extra data.

Note: Any key (in `data`) prefixed with _ will not automatically output on the Sentry details view.

Sentry will intelligently group messages if you use proper string formatting. For example, the following messages would be seen as the same message within Sentry:

```
logger.error('There was some %s error', 'crazy')
logger.error('There was some %s error', 'fun')
logger.error('There was some %s error', 1)
```

5.3.7 Pylons

Pylons is a framework for Python.

WSGI Middleware

A Pylons-specific middleware exists to enable easy configuration from settings:

```
from raven.contrib.pylons import Sentry

application = Sentry(application, config)
```

Configuration is handled via the sentry namespace:

```
[sentry]
dsn=___DSN___
include_paths=my.package,my.other.package,
exclude_paths=my.package.crud
```

Logger setup

Add the following lines to your project's *.ini* file to setup *SentryHandler*:

```
[loggers]
keys = root, sentry

[handlers]
keys = console, sentry

[formatters]
keys = generic
```

```
[logger_root]
level = INFO
handlers = console, sentry

[logger_sentry]
level = WARN
handlers = console
qualname = sentry.errors
propagate = 0

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[handler_sentry]
class = raven.handlers.logging.SentryHandler
args = ('SENTRY_DSN',)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(asctime)s,%(msecs)03d %(levelname)-5.5s [%(name)s] %(message)s
datefmt = %H:%M:%S
```

Note: You may want to setup other loggers as well.

5.3.8 Pyramid

PasteDeploy Filter

A filter factory for PasteDeploy exists to allow easily inserting Raven into a WSGI pipeline:

```
[pipeline:main]
pipeline =
    raven
    tm
    MyApp

[filter:raven]
use = egg:raven#raven
dsn = http://public:secret@example.com/1
include_paths = my.package, my.other.package
exclude_paths = my.package.crud
```

In the `[filter:raven]` section, you must specify the entry-point for raven with the `use =` key. All other raven client parameters can be included in this section as well.

See the Pyramid PasteDeploy Configuration Documentation for more information.

Logger setup

Add the following lines to your project's *.ini* file to setup *SentryHandler*:

```
[loggers]
keys = root, sentry

[handlers]
keys = console, sentry

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console, sentry

[logger_sentry]
level = WARN
handlers = console
qualname = sentry.errors
propagate = 0

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[handler_sentry]
class = raven.handlers.logging.SentryHandler
args = ('http://public:secret@example.com/1',)
level = WARNING
formatter = generic

[formatter_generic]
format = %(asctime)s,%(msecs)03d %(levelname)-5.5s [%(name)s] %(message)s
datefmt = %H:%M:%S
```

Note: You may want to setup other loggers as well. See the Pyramid Logging Documentation for more information.

5.3.9 Configuring RQ

Starting with RQ version 0.3.1, support for Sentry has been built in.

Usage

The simplest way is passing your `SENTRY_DSN` through `rqworker`:

```
$ rqworker --sentry-dsn="___DSN___"
```

Custom Client

It's possible to use a custom `Client` object and use your own worker process as an alternative to `rqworker`.

Please see `rq`'s documentation for more information: http://python-rq.org/patterns/sentry/

5.3.10 Tornado

Tornado is an async web framework for Python.

Setup

The first thing you'll need to do is to initialize sentry client under your application

```
import tornado.web
from raven.contrib.tornado import AsyncSentryClient

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

application = tornado.web.Application([
    (r"/", MainHandler),
])
application.sentry_client = AsyncSentryClient(
    '___DSN___'
)
```

Usage

Once the sentry client is attached to the application, request handlers can automatically capture uncaught exceptions by inheriting the *SentryMixin* class.

```
import tornado.web
from raven.contrib.tornado import SentryMixin

class UncaughtExceptionExampleHandler(
        SentryMixin, tornado.web.RequestHandler):
    def get(self):
        1/0
```

You can also send events manually using the shortcuts defined in *SentryMixin*. The shortcuts can be used for both asynchronous and synchronous usage.

Asynchronous

```
import tornado.web
import tornado.gen
from raven.contrib.tornado import SentryMixin

class AsyncMessageHandler(SentryMixin, tornado.web.RequestHandler):
    @tornado.web.asynchronous
    @tornado.gen.engine
    def get(self):
        self.write("You requested the main page")
        yield tornado.gen.Task(
            self.captureMessage, "Request for main page served"
        )
        self.finish()

class AsyncExceptionHandler(SentryMixin, tornado.web.RequestHandler):
```

```
    @tornado.web.asynchronous
    @tornado.gen.engine
    def get(self):
        try:
            raise ValueError()
        except Exception as e:
            response = yield tornado.gen.Task(
                self.captureException, exc_info=True
            )
        self.finish()
```

Tip: The value returned by the yield is a `HTTPResponse` object.

Synchronous

```
import tornado.web
from raven.contrib.tornado import SentryMixin

class AsyncExampleHandler(SentryMixin, tornado.web.RequestHandler):
    def get(self):
        self.write("You requested the main page")
        self.captureMessage("Request for main page served")
```

5.3.11 WSGI Middleware

Raven includes a simple to use WSGI middleware.

```
from raven import Client
from raven.middleware import Sentry

application = Sentry(
    application,
    Client('http://public:secret@example.com/1')
)
```

Note: Many frameworks will not propagate exceptions to the underlying WSGI middleware by default.

5.3.12 ZeroRPC

ZeroRPC is a light-weight, reliable and language-agnostic library for distributed communication between server-side processes.

Setup

The ZeroRPC integration comes as middleware for ZeroRPC. The middleware can be configured like the original Raven client (using keyword arguments) and registered into ZeroRPC's context manager:

```
import zerorpc

from raven.contrib.zerorpc import SentryMiddleware
```

```
sentry = SentryMiddleware(dsn='___DSN___')
zerorpc.Context.get_instance().register_middleware(sentry)
```

By default, the middleware will hide internal frames from ZeroRPC when it submits exceptions to Sentry. This behavior can be disabled by passing the `hide_zerorpc_frames` parameter to the middleware:

```
sentry = SentryMiddleware(hide_zerorpc_frames=False, dsn='___DSN___')
```

Compatibility

- ZeroRPC-Python < 0.4.0 is compatible with Raven <= 3.1.0;
- ZeroRPC-Python >= 0.4.0 requires Raven > 3.1.0.

5.3.13 Zope/Plone

zope.conf

Zope has extensible logging configuration options. A basic setup for logging looks like that:

```
<eventlog>
  level INFO
  <logfile>
   path ${buildout:directory}/var/{:_buildout_section_name_}.log
   level INFO
  </logfile>

  %import raven.contrib.zope
  <sentry>
    dsn ___DSN___
    level ERROR
  </sentry>
</eventlog>
```

This configuration keeps the regular logging to a logfile, but adds logging to sentry for ERRORs.

All options of `raven.base.Client` are supported.

Nobody writes zope.conf files these days, instead buildout recipe does that. To add the equivalent configuration, you would do this:

```
[instance]
recipe = plone.recipe.zope2instance
...
event-log-custom =
    %import raven.contrib.zope
    <logfile>
      path ${buildout:directory}/var/instance.log
      level INFO
    </logfile>
    <sentry>
      dsn ___DSN___
      level ERROR
    </sentry>
```

5.4 Transports

A transport is the mechanism in which Raven sends the HTTP request to the Sentry server. By default, Raven uses a threaded asynchronous transport, but you can easily adjust this by modifying your `SENTRY_DSN` value.

Transport registration is done via the URL prefix, so for example, a synchronous transport is as simple as prefixing your `SENTRY_DSN` with the `sync+` value.

Options are passed to transports via the querystring.

All transports should support at least the following options:

`timeout = 1` The time to wait for a response from the server, in seconds.

`verify_ssl = 1` If the connection is HTTPS, validate the certificate and hostname.

`ca_certs = [raven]/data/cacert.pem` A certificate bundle to use when validating SSL connections.

For example, to increase the timeout and to disable SSL verification:

```
SENTRY_DSN = '___DSN___?timeout=5&verify_ssl=0'
```

5.4.1 aiohttp

Should only be used within a **PEP 3156** compatible event loops (*asyncio* itself and others).

```
SENTRY_DSN = 'aiohttp+___DSN___'
```

5.4.2 Eventlet

Should only be used within an Eventlet IO loop.

```
SENTRY_DSN = 'eventlet+___DSN___'
```

5.4.3 Gevent

Should only be used within a Gevent IO loop.

```
SENTRY_DSN = 'gevent+___DSN___'
```

5.4.4 Requests

Requires the `requests` library. Synchronous.

```
SENTRY_DSN = 'requests+___DSN___'
```

5.4.5 Sync

A synchronous blocking transport.

```
SENTRY_DSN = 'sync+___DSN___'
```

5.4.6 Threaded (Default)

Spawns an async worker for processing messages.

```
SENTRY_DSN = 'threaded+___DSN___'
```

5.4.7 Tornado

Should only be used within a Tornado IO loop.

```
SENTRY_DSN = 'tornado+___DSN___'
```

5.4.8 Twisted

Should only be used within a Twisted event loop.

```
SENTRY_DSN = 'twisted+___DSN___'
```

5.5 Supported Platforms

- Python 2.6
- Python 2.7
- Python 3.2
- Python 3.3
- PyPy
- Google App Engine

5.6 API Reference

This gives you an overview of the public API that raven-python exposes.

5.6.1 Client

class raven.**Client**(*dsn=None*, ***kwargs*)

The client needs to be instanciated once and can then be used for submitting events to the Sentry server. For information about the configuration of that client and which parameters are accepted see *Configuring the Client*.

capture(*event_type*, *data=None*, *date=None*, *time_spent=None*, *extra=None*, *stack=False*, *tags=None*, ***kwargs*)

This method is the low-level method for reporting events to Sentry. It captures and processes an event and pipes it via the configured transport to Sentry.

Example:

```
        capture('raven.events.Message', message='foo', data={
            'request': {
                'url': '...',
                'data': {},
                'query_string': '...',
                'method': 'POST',
            },
            'logger': 'logger.name',
        }, extra={
            'key': 'value',
        })
```

Parameters

- **event_type** – the module path to the Event class. Builtins can use shorthand class notation and exclude the full module path.
- **data** – the data base, useful for specifying structured data interfaces. Any key which contains a '.' will be assumed to be a data interface.
- **date** – the datetime of this event. If not supplied the current timestamp is used.
- **time_spent** – a integer value representing the duration of the event (in milliseconds)
- **extra** – a dictionary of additional standard metadata.
- **stack** – If set to *True* a stack frame is recorded together with the event.
- **tags** – list of extra tags
- **kwargs** – extra keyword arguments are handled specific to the reported event type.

Returns a tuple with a 32-length string identifying this event

captureMessage(*message*, ***kwargs*)

This is a shorthand to reporting a message via *capture()*. It passes `'raven.events.Message'` as *event_type* and the message along. All other keyword arguments are regularly forwarded.

Example:

```
        client.captureMessage('This just happened!')
```

captureException(*message*, *exc_info=None*, ***kwargs*)

This is a shorthand to reporting an exception via *capture()*. It passes `'raven.events.Exception'` as *event_type* and the traceback along. All other keyword arguments are regularly forwarded.

If exc_info is not provided, or is set to True, then this method will perform the `exc_info = sys.exc_info()` and the requisite clean-up for you.

Example:

```
        try:
            1 / 0
        except Exception:
            client.captureException()
```

send(***data*)

Accepts all data parameters and serializes them, then sends then onwards via the transport to Sentry. This can be used as to send low-level protocol data to the server.

context
Returns a reference to the thread local context object. See *raven.context.Context* for more information.

5.6.2 Context

class raven.context.**Context**
The context object works similar to a dictionary and is used to record information that should be submitted with events automatically. It is available through *raven.Client.context* and is thread local. This means that you can modify this object over time to feed it with more appropriate information.

merge(*data*)
Performs a merge of the current data in the context and the new data provided.

clear()
Clears the context. It's important that you make sure to call this when you reuse the thread for something else. For instance for web frameworks it's generally a good idea to call this at the end of the HTTP request.

Otherwise you run at risk of seeing incorrect information after the first use of the thread.

CHAPTER 6

For Developers

6.1 Contributing

Want to contribute back to Sentry? This page describes the general development flow, our philosophy, the test suite, and issue tracking.

(Though it actually doesn't describe all of that, yet)

6.1.1 Setting up an Environment

Sentry is designed to run off of setuptools with minimal work. Because of this setting up a development environment requires only a few steps.

The first thing you're going to want to do, is build a virtualenv and install any base dependancies.

```
virtualenv ~/.virtualenvs/raven
source ~/.virtualenvs/raven/bin/activate
make
```

That's it :)

6.1.2 Running the Test Suite

The test suite is also powered off of py.test, and can be run in a number of ways. Usually though, you'll just want to use our helper method to make things easy:

```
make test
```

6.1.3 Contributing Back Code

Ideally all patches should be sent as a pull request on GitHub, and include tests. If you're fixing a bug or making a large change the patch **must** include test coverage.

You can see a list of open pull requests (pending changes) by visiting https://github.com/getsentry/raven-python/pulls

CHAPTER 7

Supported Platforms

- Python 2.6
- Python 2.7
- Python 3.2
- Python 3.3
- Python 3.4
- Python 3.5
- PyPy
- Google App Engine

CHAPTER 8

Deprecation Notes

Milestones releases are 1.3 or 1.4, and our deprecation policy is to a two version step. For example, a feature will be deprecated in 1.3, and completely removed in 1.4.

CHAPTER 9

Resources

- Documentation
- Bug Tracker
- Code
- Mailing List
- IRC (irc.freenode.net, #sentry)

C

M

P

R

S