Eliot Documentation

Release 1.2.0+0.g0545fd3

ClusterHQ

Oct 28, 2017

Contents

Most logging systems tell you *what* happened in your application, whereas `eliot` also tells you *why* it happened.

`eliot` is a Python logging system that outputs causal chains of **actions**: actions can spawn other actions, and eventually they either **succeed or fail**. The resulting logs tell you the story of what your software did: what happened, and what caused it.

Eliot works well within a single process, but can also be used across multiple processes to trace causality across a distributed system. Eliot is only used to generate your logs; you will still need tools like Logstash and ElasticSearch to aggregate and store logs if you are using multiple processes.

- **Start here:** *Quickstart documentation*
- Need help? File an issue or join the `#eliot` IRC channel on `irc.freenode.net`
- Read on for the full documentation.

CHAPTER 1

Testimonials

"Eliot has made tracking down causes of failure (in complex external integrations and internal uses) tremendously easier. Our errors are logged to Sentry with the Eliot task UUID. That means we can go from a Sentry notification to a high-level trace of operations—with important metadata at each operation—in a few seconds. We immediately know which user did what in which part of the system."

—Jonathan Jacobs

CHAPTER 2

Documentation

Quickstart

Let's see how easy it is to use Eliot.

Installing Eliot

To install Eliot and the other tools we'll use in this example, run the following in your shell:

```
$ pip install eliot eliot-tree requests
```

This will install:

1. Eliot itself.
2. eliot-tree, a tool that lets you visualize Eliot logs easily.
3. `requests`, a HTTP client library we'll use in the example code below. You don't need it for real Eliot usage, though.

Our example program

We're going to add logging code to the following script, which checks if a list of links are valid URLs:

```
import requests

def check_links(urls):
    for url in urls:
        try:
            response = requests.get(url)
            response.raise_for_status()
        except Exception as e:
            raise ValueError(str(e))
```

```
try:
    check_links(["http://eliot.readthedocs.io", "http://nosuchurl"])
except ValueError:
    print("Not all links were valid.")
```

Adding Eliot logging

To add logging to this program, we do two things:

1. Tell Eliot to log messages to file called "linkcheck.log" by using `eliot.to_file()`.
2. Create two actions using `eliot.start_action()`. Actions succeed when the `eliot.start_action()` context manager finishes successfully, and fail when an exception is raised.

```
import requests
from eliot import start_action, to_file
to_file(open("linkcheck.log", "w"))


def check_links(urls):
    with start_action(action_type="check_links", urls=urls):
        for url in urls:
            try:
                with start_action(action_type="download", url=url):
                    response = requests.get(url)
                    response.raise_for_status()
            except Exception as e:
                raise ValueError(str(e))

try:
    check_links(["http://eliot.readthedocs.io", "http://nosuchurl"])
except ValueError:
    print("Not all links were valid.")
```

Running the code

Let's run the code:

```
$ python linkcheck.py
Not all the links were valid.
```

We can see the resulting log file is composed of JSON messages, one per line:

```
$ cat linkcheck.log
{"action_status": "started", "task_uuid": "b1cb58cf-2c2f-45c0-92b2-838ac00b20cc",
↪"task_level": [1], "timestamp": 1509136967.2066844, "action_type": "check_links",
↪"urls": ["http://eliot.readthedocs.io", "http://nosuchurl"]}
...
```

So far these logs seem similar to the output of regular logging systems: individual isolated messages. But unlike those logging systems, Eliot produces logs that can be reconstructed into a tree, for example using the `eliot-tree` utility:

```
$ eliot-tree linkcheck.log
b1cb58cf-2c2f-45c0-92b2-838ac00b20cc
```

```
- check_links/1  started
    - timestamp: 2017-10-27 20:42:47.206684
    - urls:
    |   - 0: http://eliot.readthedocs.io
    |   - 1: http://nosuchurl
    - download/2/1  started
    |   - timestamp: 2017-10-27 20:42:47.206933
    |   - url: http://eliot.readthedocs.io
    |   - download/2/2  succeeded
    |       - timestamp: 2017-10-27 20:42:47.439203
    - download/3/1  started
    |   - timestamp: 2017-10-27 20:42:47.439412
    |   - url: http://nosuchurl
    |   - download/3/2  failed
    |       - errno: None
    |       - exception: requests.exceptions.ConnectionError
    |       - reason: HTTPConnectionPool(host='nosuchurl', port=80): Max retries␣
↪exceeded with url: / (Caused by NewConnec...
    |       - timestamp: 2017-10-27 20:42:47.457133
    - check_links/4  failed
        - exception: builtins.ValueError
        - reason: HTTPConnectionPool(host='nosuchurl', port=80): Max retries exceeded␣
↪with url: / (Caused by NewConnec...
        - timestamp: 2017-10-27 20:42:47.457332
```

Notice how:

1. Eliot tells you which actions succeeded and which failed.
2. Failed actions record their exceptions.
3. You can see just from the logs that the `check_links` action caused the `download` action.

Next steps

You can learn more by reading the rest of the documentation, including:

- The *motivation behind Eliot*.
- How to generate *actions*, *standalone messages*, and *handle errors*.
- How to integrate with *asyncio coroutines*, *threads and processes*, or *Twisted*.
- How to output logs *to a file or elsewhere*.

Why Eliot?

> Suppose we turn from outside estimates of a man, to wonder, with keener interest, what is the report of his own consciousness about his doings or capacity: with what hindrances he is carrying on his daily labors; what fading of hopes, or what deeper fixity of self-delusion the years are marking off within him; and with what spirit he wrestles against universal pressure, which will one day be too heavy for him, and bring his heart to its final pause.
>
> — George Eliot, *Middlemarch*

The log messages generated by a piece of software tell a story: what, where, when, even why and how if you're lucky. The readers of this story are more often than not other programs: monitoring systems, performance tools, or

just filtering the messages down to something a human can actually comprehend. Unfortunately the output of most logging systems is ill-suited to being read by programs. Even worse, most logging systems omit critical information that both humans and their programs need.

Problem #1: Text is hard to search

Let's say you want to find all the log messages about a specific person. A first pass of log messages might look like this:

> Sir James Chettam was going to dine at the Grange to-day with another gentleman whom the girls had never seen, and about whom Dorothea felt some venerating expectation. ... If Miss Brooke ever attained perfect meekness, it would not be for lack of inward fire.

You could do a text search for log messages containing the text "Dorothea", but this is likely to fail for some types of searches. You might want to searching for actions involving dinner, but then you would need to search for "dine" and "dinner" and perhaps other words well. A library like structlog that can generate structured log messages will solve this first problem. You could define a "person" field in your messages and then you can search for all messages where `person == "Dorothea"` as well as other structured queries.

Problem #2: Referring to Entities

Every time a log message is written out you need to decide how to refer to the objects being logged. In the messages we saw above "Dorothea" and "Miss Brooke" are in fact different identifiers for the same person. Having structured messages doesn't help us find all messages about a specific entity if the object is referred to inconsistently. What you need is infrastructure for converting specific kinds of objects in your code to fields in your structured log messages. Then you can just say "log a message that refers to this Person" and that reusable code will make sure the correct identifier is generated.

Problem #3: Actions

Most log messages in your program are going to involve actions:

> Not long after that dinner-party she had become Mrs. Casaubon, and was on her way to Rome.

A marriage has a beginning and eventually an end. The end may be successful, presuming "until death do us part" is a form of success, or a failure. The same is true of all actions, much like function calls in Python are started and eventually return a result or throw an exception. Actions may of course span multiple function calls or extended periods of time.

Actions also generate other actions: a marriage leads to a trip to Rome, the trip to Rome might lead to a visit to the Vatican Museum, and so on. Other unrelated actions are occurring at the same time, resulting in a forest of actions, with root actions that grow a tree of child actions.

You might want to trace an action from beginning to end, e.g. to measure how long it took to run. You might want to know what high-level action caused a particular unexpected low-level action. You might want to know what actions a specific entity was involved with. None of these are possible in most logging systems since they have no concept of actions to begin with.

Problem #4: Cross-Process Actions

A story may involve many characters in many places at many times. The novelist has the power to present the internal consciousness of not just one character but many: their ways of thinking, their different perceptions of reality.

Similarly, actions in a distributed system may span multiple processes. An incoming request to one server may cause a ripple of effects reaching many other processes; the logs from a single process in isolation are insufficient to understand what happened and why.

The Solution: Eliot

Eliot is designed to solve all of these problems. For simplicity's sake this example focuses on problems 1 and 3; problem 2 is covered by the *type system* and problem 4 by *cross-process actions*.

```
from sys import stdout
from eliot import Message, to_file
to_file(stdout)


class Place(object):
    def __init__(self, name, contained=()):
        self.name = name
        self.contained = contained

    def visited(self, people):
        Message.log(message_type="visited",
                    people=people, place=self.name)
        for thing in self.contained:
            thing.visited(people)


def honeymoon(family, destination):
    Message.log(message_type="honeymoon", people=family)
    destination.visited(family)


honeymoon(["Mrs. Casaubon", "Mr. Casaubon"],
          Place("Rome, Italy",
                [Place("Vatican Museum",
                       [Place("Statue #1"), Place("Statue #2")])]))
```

Here's how the log messages generated by the code look, as summarized by the eliot-tree tool:

```
68c12428-5d60-49f5-a269-3fb681938f98
+-- honeymoon@1
    |-- people: [u'Mrs. Casaubon', u'Mr. Casaubon']

361298ae-b6b7-439a-bc9b-ffde68b7860d
+-- visited@1
    |-- people: [u'Mrs. Casaubon', u'Mr. Casaubon']
    |-- place: Rome, Italy

7fe1615c-e442-4bca-b667-7bb435ac6cb8
+-- visited@1
    |-- people: [u'Mrs. Casaubon', u'Mr. Casaubon']
    |-- place: Vatican Museum

c746230c-627e-4ff9-9173-135568df976c
+-- visited@1
    |-- people: [u'Mrs. Casaubon', u'Mr. Casaubon']
    |-- place: Statue #1
```

```
5482ec10-36c6-4194-964f-074e325b9329
+-- visited@1
    |-- people: [u'Mrs. Casaubon', u'Mr. Casaubon']
    |-- place: Statue #2
```

We can see different messages are related insofar as they refer to the same person, or the same thing... but we can't trace the relationship in terms of actions. Was looking at a statue the result of visiting Rome? There's no way we can tell from the log messages. We could manually log start and finish messages but that won't suffice when we have many interleaved actions involving the same objects. Which of twenty parallel HTTP request tried to insert a row into the database? Chronological messages simply cannot tell us that.

The solution is to introduce two new concepts: actions and tasks. An "action" is something with a start and an end; the end can be successful or it can fail due to an exception. Log messages, as well as log actions, know the log action whose context they are running in. The result is a tree of actions. A "task" is a top-level action, a basic entry point into the program which drives other actions. The task is therefore the root of the tree of actions. For example, an HTTP request received by a web server might be a task.

In our example we have one task (the honeymoon), an action (travel). We will leave looking as a normal log message because it always succeeds, and no other log message will ever need to run its context. Here's how our code looks now:

```
from sys import stdout
from eliot import start_action, start_task, to_file
to_file(stdout)


class Place(object):
    def __init__(self, name, contained=()):
        self.name = name
        self.contained = contained

    def visited(self, people):
        # No need to repetitively log people, since caller will:
        with start_action(action_type="visited", place=self.name):
            for thing in self.contained:
                thing.visited(people)


def honeymoon(family, destination):
    with start_task(action_type="honeymoon", people=family):
        destination.visited(family)


honeymoon(["Mrs. Casaubon", "Mr. Casaubon"],
          Place("Rome, Italy",
                [Place("Vatican Museum",
                       [Place("Statue #1"), Place("Statue #2")])]))
```

Actions provide a Python context manager. When the action or task starts a start message is logged. If the block finishes successfully a success message is logged for the action; if an exception is thrown a failure message is logged for the action with the exception type and contents. Not shown here but supported by the API is the ability to add fields to the success messages for an action. A similar API supports Twisted's Deferreds.

Here's how the log messages generated by the new code look, as summarized by the eliot-tree tool:

```
f9dcc74f-ecda-4543-9e9a-1bb062d199f0
+-- honeymoon@1/started
    |-- people: [u'Mrs. Casaubon', u'Mr. Casaubon']
```

```
+-- visited@2,1/started
    |-- place: Rome, Italy
    +-- visited@2,2,1/started
        |-- place: Vatican Museum
        +-- visited@2,2,2,1/started
            |-- place: Statue #1
            +-- visited@2,2,2,2/succeeded
        +-- visited@2,2,3,1/started
            |-- place: Statue #2
            +-- visited@2,2,3,2/succeeded
        +-- visited@2,2,4/succeeded
    +-- visited@2,3/succeeded
+-- honeymoon@3/succeeded
```

No longer isolated fragments of meaning, our log messages are now a story. Log events have context, you can tell where they came from and what they led to without guesswork. Was looking at a statue the result of the honeymoon? It most definitely was.

What's New

1.2.0

Features:

- Eliot now does the right thing for `asyncio` coroutines in Python 3.5 and later. See *Asyncio Coroutine Support* for details.

Misc:

- `Action.continue_task` can now accept text task IDs (`str` in Python 3, `unicode` in Python 2).

1.1.0

Features:

- Messages are no longer lost if they are logged before any destinations are added. In particular, messages will be buffered in memory until the first set of destinations are added, at which point those messages will be delivered. Thanks to Jean-Paul Calderone for the feature request.
- `eliot.add_destinations` replaces `eliot.add_destination`, and accepts multiple Destinations at once.
- `eliot.twisted.TwistedDestination` allows redirecting Eliot logs to `twisted.logger`. Thanks to Glyph Lefkowitz for the feature request.

Misc:

- Coding standard switched to PEP-8.
- Dropped support for Python 3.3.
- Dropped support for versions of Twisted older than 15.2 (or whenever it was that `twisted.logger` was introduced).
- Dropped support for `ujson`.

1.0.0

Eliot is stable, and has been for a while, so switching to v1.0.

Features:

- New API: `MessageType.log()`, the equivalent of `Message.log()`, allows you to quickly create a new typed log message and write it out.
- New APIs: `eliot.current_action()` returns the current `Action`, and `Action.task_uuid` is the task's UUID.
- You can now do `with YOUR_ACTION().context() as action:`, i.e. `Action.context()` context manager returns the `Action` instance.
- `ActionType.as_task` no longer requires a logger argument, matching the other APIs where passing in a logger is optional.

0.12.0

Features:

- Python 3.6 support.

Misc:

- Made test suite pass again with latest Hypothesis release.

0.11.0

Features:

- Eliot tasks can now more easily *span multiple threads* using the new `eliot.preserve_context` API.
- `eliot-prettyprint` command line tool now pretty prints field values in a more informative manner.

Bug fixes:

- `eliot-prettyprint` now handles unparseable lines by skipping formatting them rather than exiting.

0.10.1

Bug fixes:

- Fixed regression in 0.10.0: fix validation of failed actions and tracebacks with extracted additional fields.

0.10.0

Features:

- `register_exception_extractor` allows for more useful *logging of failed actions and tracebacks* by extracting additional fields from exceptions.
- Python 3.5 support.

Bug fixes:

- Journald support works on Python 3.

0.9.0

Features:

- Native *journald support*.
- `eliot-prettyprint` is a command-line tool that formats JSON Eliot messages into a more human-friendly format.
- `eliot.logwriter.ThreadedWriter` is a Twisted non-blocking wrapper for any blocking destination.

0.8.0

Features:

- `Message.log` will log a new message, combining the existing `Message.new` and `Message.write`.
- `write_traceback` and `writeFailure` no longer require a `Logger`; they now default to using the global one.
- The logs written with `redirectLogsForTrial` are now written in JSON format, rather than with `pformat`.

Bug fixes:

- `FileDestination` will now call `flush()` on the given file object after writing the log message. Previously log messages would not end up being written out until the file buffer filled up.
- Each `Message` logged outside the context of an action now gets a unique `task_id`.

0.7.0

- Creating your own `Logger` instances is no longer necessary; all relevant APIs now default to using a global one. A new testing decorator (`eliot.testing.capture_logging`) was added to capture global logging.
- Support positional `Field`-instance arguments to `fields()` to make combining existing field types and simple fields more convenient. Contributed by Jonathan Jacobs.
- `write_traceback` and `writeFailure` no longer require a `system` argument, as the combination of traceback and action context should suffice to discover the origin of the problem. This is a minor change to output format as the field is also omitted from the resulting `eliot:traceback` messages.
- The `validate_logging` testing utility now skips validation when the decorated test method raises `SkipTest`.
- Exceptions in destinations are now handled better: instead of being dropped silently an attempt is made to log a message about the problem. If that also fails then the exception is dropped.

0.6.0

Warning: Incompatible output format change! In previous versions the ordering of messages and actions was ambiguous and could not be deduced from out-of-order logs, and even where it was possible sorting correctly was difficult. To fix this the `action_counter` field was removed and now all messages can be uniquely located within a specific task by the values in an *improved task_level field*.

Features:

- Eliot tasks can now *span multiple processes and threads*, allowing for easy tracing of actions in complex and distributed applications.
- *eliot.add_global_fields* allows adding fields with specific values to all Eliot messages logged by your program. This can be used to e.g. distinguish between log messages from different processes by including relevant identifying information.

Bug fixes:

- On Python 3 files that accept unicode (e.g. `sys.stdout`) should now work.

0.5.0

Features:

- Added support for Python 3.4.
- Most public methods and functions now have underscore-based equivalents to the camel case versions, e.g. `eliot.write_traceback` and `eliot.writeTraceback`, for use in PEP 8 styled programs. Twisted-facing APIs and pyunit assertions do not provide these additional APIs, as camel-case is the native idiom.
- `eliot.to_file` outputs log messages to a file.
- Documented how to load Eliot logging into ElasticSearch via Logstash.
- Documentation has been significantly reorganized.

0.4.0

Note that this is the last release that will make incompatible API changes without interim deprecation warnings.

Incompatible changes from 0.3.0:

- `Logger` no longer does JSON serialization; it's up to destinations to decide how to serialize the dictionaries they receive.
- Timestamps are no longer encoded in TAI64N format; they are now provided as seconds since the Unix epoch.
- `ActionType` no longer supports defining additional failure fields, and therefore accepts one argument less.
- `Action.runCallback` and `Action.finishAfter` have been removed, as they are replaced by `DeferredContext` (see below).

Features:

- Added a simpler API (`fields()`) for defining fields for `ActionType` and `MessageType`.
- Added support for Python 3.3.
- Actions can now be explicitly finished using a public API: `Action.finish()`.
- `Action.context()` context manager allows setting an action context without finishing the action when exiting the block.
- Added a new API for Twisted `Deferred` support: `eliot.twisted.DeferredContext`.
- `eliot.twisted.redirectLogsForTrial` will redirect Eliot logs to Twisted's logs when running under the `trial` test runner.

Generating Logs

Messages

Basic usage

At its base, Eliot outputs structured messages composed of named fields. Eliot messages are typically serialized to JSON objects. Fields therefore can have Unicode names, so either `unicode` or `bytes` containing UTF-8 encoded Unicode. Message values must be supported by JSON: `int`, `float`, `None`, `unicode`, UTF-8 encoded Unicode as `bytes`, `dict` or `list`. The latter two can only be composed of other supported types.

You can log a message like this:

```
from eliot import Message

class YourClass(object):
    def run(self):
        # Log a message with two fields, "key" and "value":
        Message.log(key=123, value=u"hello")
```

You can also create message and then log it later like this:

```
from eliot import Message

class YourClass(object):
    def run(self):
        # Create a message with two fields, "key" and "value":
        msg = Message.new(key=123, value=u"hello")
        # Write the message:
        msg.write()
```

Message binding

You can also create a new `Message` from an existing one by binding new values. New values will override ones on the base `Message`, but `bind()` does not mutate the original `Message`.

```
# This message has fields key=123, value=u"hello"
msg = Message.new(key=123, value=u"hello")
# And this one has fields key=123, value=u"other", extra=456
msg2 = msg.bind(value=u"other", extra=456)
```

Actions and Tasks

Actions: A Start and a Finish

A higher-level construct than messages is the concept of an action. An action can be started, and then finishes either successfully or with some sort of an exception. Success in this case simply means no exception was thrown; the result of an action may be a successful response saying "this did not work". Log messages are emitted for action start and finish.

Actions are also nested; one action can be the parent of another. An action's parent is deduced from the Python call stack and context managers like `Action.context()`. Log messages will also note the action they are part of if

they can deduce it from the call stack. The result of all this is that you can trace the operation of your code as it logs various actions, and see a narrative of what happened and what caused it to happen.

Logging Actions

Here's a basic example of logging an action:

```
from eliot import start_action

with start_action(action_type=u"store_data"):
    x = get_data()
    store_data(x)
```

This will log an action start message and if the block finishes successfully an action success message. If an exception is thrown by the block then an action failure message will be logged along with the exception type and reason as additional fields. Each action thus results in two messages being logged: at the start and finish of the action. No traceback will be logged so if you want a traceback you will need to do so explicitly. Notice that the action has a name, with a subsystem prefix. Again, this should be a logical name.

Note that all code called within this block is within the context of this action. While running the block of code within the `with` statement new actions created with `start_action` will get the top-level `start_action` as their parent.

Tasks: Top-level Actions

A top-level action with no parent is called a task, the root cause of all its child actions. E.g. a web server receiving a new HTTP request would create a task for that new request. Log messages emitted from Eliot are therefore logically structured as a forest: trees of actions with tasks at the root. If you want to ignore the context and create a top-level task you can use the `eliot.start_task` API.

From Actions to Messages

While the logical structure of log messages is a forest of actions, the actual output is effectively a list of dictionaries (e.g. a series of JSON messages written to a file). To bridge the gap between the two structures each output message contains special fields expressing the logical relationship between it and other messages:

- `task_uuid`: The unique identifier of the task (top-level action) the message is part of.
- `task_level`: The specific location of this message within the task's tree of actions. For example, `[3, 2, 4]` indicates the message is the 4th child of the 2nd child of the 3rd child of the task.

Consider the following code sample:

```
from eliot import start_action, start_task, Message

with start_task(action_type="parent"):
    Message.log(message_type="info", x=1)
    with start_action(action_type="child"):
        Message.log(message_type="info", x=2)
    raise RuntimeError("ono")
```

All these messages will share the same UUID in their `task_uuid` field, since they are all part of the same high-level task. If you sort the resulting messages by their `task_level` you will get the tree of messages:

```
task_level=[1] action_type="parent" action_status="started"
task_level=[2] message_type="info" x=1
    task_level=[3, 1] action_type="child" action_status="started"
    task_level=[3, 2] message_type="info" x=2
    task_level=[3, 3] action_type="child" action_status="succeeded"
task_level=[4] action_type="parent" action_status="failed" exception="exceptions.
↪RuntimeError" reason="ono"
```

Non-Finishing Contexts

Sometimes you want to have the action be the context for other messages but not finish automatically when the block finishes. You can do so with `Action.context()`. You can explicitly finish an action by calling `eliot.Action.finish`. If called with an exception it indicates the action finished unsuccessfully. If called with no arguments it indicates that the action finished successfully. Keep in mind that code within the context block that is run after the action is finished will still be in that action's context.

```
from eliot import start_action

action = start_action(action_type=u"yourapp:subsystem:frob")
try:
    with action.context():
        x = _beep()
    with action.context():
        frobinate(x)
    # Action still isn't finished, need to so explicitly.
except FrobError as e:
    action.finish(e)
else:
    action.finish()
```

The `context()` method returns the `Action`:

```
from eliot import start_action

with start_action(action_type=u"your_type").context() as action:
    # do some stuff...
    action.finish()
```

You can also explicitly run a function within the action context:

```
from eliot import start_action

action = start_action(action_type=u"yourapp:subsystem:frob")
# Call do_something(x=1) in context of action, return its result:
result = action.run(do_something, x=1)
```

Action Fields

You can add fields to both the start message and the success message of an action.

```
from eliot import start_action

with start_action(action_type=u"yourapp:subsystem:frob",
                  # Fields added to start message only:
```

```
                key=123, foo=u"bar") as action:
    x = _beep(123)
    result = frobinate(x)
    # Fields added to success message only:
    action.add_success_fields(result=result)
```

If you want to include some extra information in case of failures beyond the exception you can always log a regular message with that information. Since the message will be recorded inside the context of the action its information will be clearly tied to the result of the action by the person (or code!) reading the logs later on.

Getting the Current Action

Sometimes it can be useful to get the current action. For example, you might want to record the current task UUID for future reference, in a bug report for example. You might also want to pass around the `Action` explicitly, rather than relying on the implicit context.

You can get the current `Action` by calling `eliot.current_action()`. For example:

```
from eliot import current_action

def get_current_uuid():
    return current_action().task_uuid
```

Errors and Exceptions

Exceptions and Tracebacks

If you are using actions you don't need to do anything special to log exceptions: if an exception is thrown within the context of an action and not caught, the action will be marked as failed and the exception will be logged with it.

If you get a completely unexpected exception you may wish to log a traceback to aid debugging:

```
from eliot import write_traceback

class YourClass(object):

    def run(self):
        try:
            dosomething()
        except:
            write_traceback()
```

Custom Exception Logging

By default both failed actions and tracebacks log the class and string-representation of the logged exception. You can add additional fields to these messages by registering a callable that converts exceptions into fields. If no extraction function is registered for a class Eliot will look for registered functions for the exception's base classes.

For example, the following registration means all failed actions that fail with a `MyException` will have a `code` field in the action end message, as will tracebacks logged with this exception:

```
class MyException(Exception):
    def __init__(self, code):
        self.code = code
```

```
from eliot import register_exception_extractor
register_exception_extractor(MyException, lambda e: {"code": e.code})
```

By default Eliot will automatically extract fields from `OSError`, `IOError` and other subclasses of Python's `EnvironmentError`.

Spanning Processes and Threads

Introduction

In many applications we are interested in tasks that exist in more than just a single thread or in a single process. For example, one server may send a request to another server over a network and we would like to trace the combined operation across both servers' logs. To make this as easy as possible Eliot supports serializing task identifiers for transfer over the network (or between threads), allowing tasks to span multiple processes.

Cross-Thread Tasks

To trace actions across threads Eliot provides the `eliot.preserve_context` API. It takes a callable that is about to be passed to a thread constructor and preserves the current Eliot context, returning a new callable. This new callable should only be used, in the thread where it will run; it will restore the Eliot context and run the original function inside of it. For example:

```
#!/usr/bin/env python

"""
Example of an Eliot action context spanning multiple threads.
"""

from __future__ import unicode_literals

from threading import Thread
from sys import stdout

from eliot import to_file, preserve_context, start_action
to_file(stdout)


def add_in_thread(x, y):
    with start_action(action_type="in_thread", x=x, y=y) as context:
        context.add_success_fields(result=x+y)


with start_action(action_type="main_thread"):
    # Preserve Eliot context and restore in new thread:
    thread = Thread(target=preserve_context(add_in_thread),
                    kwargs={"x": 3, "y": 4})
    thread.start()
    # Wait for the thread to exit:
    thread.join()
```

Here's what the result is when run:

```
$ python examples/cross_thread.py | eliot-tree
11a85c42-a13f-491c-ad44-c48b2efad0e3
+-- main_thread@1/started
    +-- eliot:remote_task@2,1/started
        +-- in_thread@2,2,1/started
            |-- x: 3
            `-- y: 4
            +-- in_thread@2,2,2/succeeded
                |-- result: 7
        +-- eliot:remote_task@2,3/succeeded
    +-- main_thread@3/succeeded
```

Cross-Process Tasks

`eliot.Action.serialize_task_id()` can be used to create some `bytes` identifying a particular location within a task. `eliot.Action.continue_task()` converts a serialized task identifier into an `eliot.Action` and then starts the `Action`. The process which created the task serializes the task identifier and sends it over the network to the process which will continue the task. This second process deserializes the identifier and uses it as a context for its own messages.

In the following example the task identifier is added as a header to a HTTP request:

```
"""
Cross-process log tracing: HTTP client.
"""
from __future__ import unicode_literals

import sys
import requests

from eliot import to_file, start_action, add_global_fields
add_global_fields(process="client")
to_file(sys.stdout)


def remote_divide(x, y):
    with start_action(action_type="http_request", x=x, y=y) as action:
        task_id = action.serialize_task_id()
        response = requests.get(
            "http://localhost:5000/?x={}&y={}".format(x, y),
            headers={"x-eliot-task-id": task_id})
        response.raise_for_status()  # ensure this is a successful response
        result = float(response.text)
        action.add_success_fields(result=result)
        return result


if __name__ == '__main__':
    with start_action(action_type="main"):
        remote_divide(int(sys.argv[1]), int(sys.argv[2]))
```

The server that receives the request then extracts the identifier:

```
"""
Cross-process log tracing: HTTP server.
"""
```

```
from __future__ import unicode_literals

import sys
from flask import Flask, request

from eliot import to_file, Action, start_action, add_global_fields
add_global_fields(process="server")
to_file(sys.stdout)


app = Flask("server")


def divide(x, y):
    with start_action(action_type="divide", x=x, y=y) as action:
        result = x / y
        action.add_success_fields(result=result)
        return result


@app.route("/")
def main():
    with Action.continue_task(task_id=request.headers["x-eliot-task-id"]):
        x = int(request.args["x"])
        y = int(request.args["y"])
        return str(divide(x, y))


if __name__ == '__main__':
    app.run()
```

Tracing logs across multiple processes makes debugging problems dramatically easier. For example, let's run the following:

```
$ python examples/cross_process_server.py > server.log
$ python examples/cross_process_client.py 5 0 > client.log
```

Here are the resulting combined logs, as visualized by eliot-tree. The reason the client received a 500 error code is completely obvious in these logs:

```
$ cat client.log server.log | eliot-tree
1e0be9be-ae56-49ef-9bce-60e850a7db09
+-- main@1/started
    |-- process: client
    +-- http_request@2,1/started
        |-- process: client
        |-- x: 3
        `-- y: 0
        +-- eliot:remote_task@2,2,1/started
            |-- process: server
            +-- divide@2,2,2,1/started
                |-- process: server
                |-- x: 3
                `-- y: 0
                +-- divide@2,2,2,2/failed
                    |-- exception: exceptions.ZeroDivisionError
                    |-- process: server
                    |-- reason: integer division or modulo by zero
```

```
            +-- eliot:remote_task@2,2,3/failed
                |-- exception: exceptions.ZeroDivisionError
                |-- process: server
                |-- reason: integer division or modulo by zero
        +-- http_request@2,3/failed
            |-- exception: requests.exceptions.HTTPError
            |-- process: client
            |-- reason: 500 Server Error: INTERNAL SERVER ERROR
    +-- main@3/failed
        |-- exception: requests.exceptions.HTTPError
        |-- process: client
        |-- reason: 500 Server Error: INTERNAL SERVER ERROR
```

Cross-Thread Tasks

`eliot.Action` objects should only be used on the thread that created them. If you want your task to span multiple threads use the API described above.

Ensuring Message Uniqueness

Serialized task identifiers should be used at most once. For example, every time a remote operation is retried a new call to `serialize_task_id()` should be made to create a new identifier. Otherwise there is a chance that you will end up with messages that have duplicate identification (i.e. two messages with matching `task_uuid` and `task_level` values), making it more difficult to trace causality.

If this is not possible you may wish to start a new Eliot task upon receiving a remote request, while still making sure to log the serialized remote task identifier. The inclusion of the remote task identifier will allow manual or automated reconstruction of the cross-process relationship between the original and new tasks.

Another alternative in some cases is to rely on unique process or thread identity to distinguish between the log messages. For example if the same serialized task identifier is sent to multiple processes, log messages within the task can still have a unique identity if a process identifier is included with each message.

Logging Output for Multiple Processes

If logs are being combined from multiple processes an identifier indicating the originating process should be included in log messages. This can be done a number of ways, e.g.:

- Have your destination add another field to the output.
- Rely on Logstash, or whatever your logging pipeline tool is, to add a field when shipping the logs to your centralized log store.

Using Types to Structure Messages and Actions

Why Typing?

So far we've been creating messages and actions in an unstructured manner. This means it's harder to support Python objects that aren't built-in and to validate message structure. Moreover there's no documentation of what fields messages and action messages expect. To improve this we introduce the preferred API for creating actions and standalone messages: `ActionType` and `MessageType`. Here's an example demonstrating how we create a message type, bind some values and then log the message:

```
from eliot import Field, MessageType

class Coordinate(object):
    def __init__(self, x, y):
        self.x = self.x
        self.y = self.y


# This field takes a complex type that will be stored in a single Field,
# so we pass in a serializer function that converts it to a list with two
# ints:
_LOCATION = Field(u"location", lambda loc: [loc.x, loc.y], u"The location.")
# These fields are just basic supported types, in this case int and unicode
# respectively:
_COUNT = Field.for_types(u"count", [int], u"The number of items to deliver.")
_NAME = Field.for_types(u"name", [unicode], u"The name of the delivery person.")

# This is a type definition for a message. It is used to hook up
# serialization of field values, and for message validation in unit tests:
LOG_DELIVERY_SCHEDULED = MessageType(
    u"pizzadelivery:schedule",
    [_LOCATION, _COUNT, _NAME],
    u"A pizza delivery has been scheduled.")


def deliver_pizzas(deliveries):
    person = get_free_delivery_person()
    for location, count in deliveries:
        delivery_database.insert(person, location, count)
        LOG_DELIVERY_SCHEDULED.log(
            name=person.name, count=count, location=location)
```

Fields

A `Field` instance is used to validate fields of messages, and to serialize rich types to the built-in supported types. It is created with the name of the field, a a serialization function that converts the input to an output and a description. The serialization function must return a result that is JSON-encodable. You can also pass in an extra validation function. If you pass this function in it will be called with values that are being validated; if it raises `eliot.ValidationError` that value will fail validation.

A couple of utility functions allow creating specific types of `Field` instances. `Field.for_value` returns a `Field` that only can have a single value. More generally useful, `Field.for_types` returns a `Field` that can only be one of certain specific types: some subset of `unicode`, `bytes`, `int`, `float`, `bool`, `list` and `dict` as well as `None` which technically isn't a class. As always, `bytes` must only contain UTF-8 encoded Unicode.

```
from eliot import Field

def userToUsername(user):
    """
    Extract username from a User object.
    """
    return user.username

USERNAME = Field(u"username", userToUsername, u"The name of the user.")

# Validation is useful for unit tests and catching bugs; it's not used in
```

```
# the actual logging code path. We therefore don't bother catching things
# we'd do in e.g. web form validation.
def _validateAge(value):
    if value is not None and value < 0:
        raise ValidationError("Field 'age' must be positive:", value)
AGE = Field.for_types(u"age", [int, None],
                      u"The age of the user, might be None if unknown",
                      _validateAge)
```

Message Types

Now that you have some fields you can create a custom `MessageType`. This takes a message name which will be put in the `message_type` field of resulting messages. It also takes a list of `Field` instances and a description.

```
from eliot import MessageType, Field
USERNAME = Field.for_types("username", [str])
AGE = Field.for_types("age", [int])

LOG_USER_REGISTRATION = MessageType(u"yourapp:authentication:registration",
                                    [USERNAME, AGE],
                                    u"We've just registered a new user.")
```

Since this syntax is rather verbose a utility function called `fields` is provided which creates a `list` of `Field` instances for you, with support to specifying the types of the fields. The equivalent to the code above is:

```
from eliot import MessageType, fields

LOG_USER_REGISTRATION = MessageType(u"yourapp:authentication:registration",
                                    fields(username=str, age=int))
```

Or you can even use existing `Field` instances with `fields`:

```
from eliot import MessageType, Field, fields

USERNAME = Field.for_types("username", [str])

LOG_USER_REGISTRATION = MessageType(u"yourapp:authentication:registration",
                                    fields(USERNAME, age=int))
```

Given a `MessageType` you can create a `Message` instance with the `message_type` field pre-populated by calling the type. You can then use it the way you would normally use `Message`, e.g. `bind()` or `write()`. You can also just call `MessageType.log()` to write out a message directly:

```
# Simple version:
LOG_USER_REGISTRATION.log(username=user, age=193)
# Equivalent more complex API:
LOG_USER_REGISTRATION(username=user).bind(age=193).write()
```

A `Message` created from a `MessageType` will automatically use the `MessageType` `Field` instances to serialize its fields.

Keep in mind that no validation is done when messages are created. Instead, validation is intended to be done in your unit tests. If you're not unit testing all your log messages you're doing it wrong. Luckily, Eliot makes it pretty easy to test logging as we'll see in a bit.

Action Types

Similarly to `MessageType` you can also create types for actions. Unlike a `MessageType` you need two sets of fields: one for action start, one for success.

```
from eliot import ActionType, fields

LOG_USER_SIGNIN = ActionType(u"yourapp:authentication:signin",
                             # Start message fields:
                             fields(username=str),
                             # Success message fields:
                             fields(status=int),
                             # Description:
                             u"A user is attempting to sign in.")
```

Calling the resulting instance is equivalent to `start_action`. For `start_task` you can call `LOG_USER_SIGNIN.as_task`.

```
def signin(user, password):
     with LOG_USER_SIGNIN(username=user) as action:
         status = user.authenticate(password)
         action.add_success_fields(status=status)
     return status
```

Again, as with `MessageType`, field values will be serialized using the `Field` definitions in the `ActionType`.

Serialization Errors

While validation of field values typically only happens when unit testing, serialization must run in the normal logging code path. Eliot tries to very hard never to raise exceptions from the log writing code path so as not to prevent actual code from running. If a message fails to serialize then a `eliot:traceback` message will be logged, along with a `eliot:serialization_failure` message with an attempt at showing the message that failed to serialize.

```
{"exception": "exceptions.ValueError",
 "timestamp": "2013-11-22T14:16:51.386745Z",
 "traceback": "Traceback (most recent call last):\n  ... ValueError: invalid literal␣
↪for int() with base 10: 'hello'\n",
 "system": "eliot:output",
 "reason": "invalid literal for int() with base 10: 'hello'",
 "message_type": "eliot:traceback"}
{"timestamp": "2013-11-22T14:16:51.386827Z",
 "message": "{u\"u'message_type'\": u\"'test'\", u\"u'field'\": u\"'hello'\", u\"u
↪'timestamp'\": u\"'2013-11-22T14:16:51.386634Z'\"}",
 "message_type": "eliot:serialization_failure"}
```

Unit Testing Your Logging

Validate Logging in Tests

Now that you've got some code emitting log messages (or even better, before you've written the code) you can write unit tests to verify it. Given good test coverage all code branches should already be covered by tests unrelated to logging. Logging can be considered just another aspect of testing those code branches. Rather than recreating all those tests as separate functions Eliot provides a decorator the allows adding logging assertions to existing tests.

Let's unit test some code that relies on the `LOG_USER_REGISTRATION` object we created earlier.

```
from myapp.logtypes import LOG_USER_REGISTRATION

class UserRegistration(object):

    def __init__(self):
        self.db = {}

    def register(self, username, password, age):
        self.db[username] = (password, age)
        LOG_USER_REGISTRATION(
             username=username, password=password, age=age).write()
```

Here's how we'd test it:

```
from unittest import TestCase
from eliot import MemoryLogger
from eliot.testing import assertContainsFields, capture_logging

from myapp.registration import UserRegistration
from myapp.logtypes import LOG_USER_REGISTRATION


class LoggingTests(TestCase):
    def assertRegistrationLogging(self, logger):
        """
        Logging assertions for test_registration.
        """
        self.assertEqual(len(logger.messages), 1)
        msg = logger.messages[0]
        assertContainsFields(self, msg,
                             {u"username": u"john",
                              u"password": u"password",
                              u"age": 12}))

    @capture_logging(assertRegistrationLogging)
    def test_registration(self, logger):
        """
        Registration adds entries to the in-memory database.
        """
        registry = UserRegistration()
        registry.register(u"john", u"password", 12)
        self.assertEqual(registry.db[u"john"], (u"passsword", 12))
```

Besides calling the given validation function the `@capture_logging` decorator will also validate the logged messages after the test is done. E.g. it will make sure they are JSON encodable. Messages were created using `ActionType` and `MessageType` will be validated using the applicable `Field` definitions. You can also call `MemoryLogger.validate` yourself to validate written messages. If you don't want any additional logging assertions you can decorate your test function using `@capture_logging(None)`.

Testing Tracebacks

Tests decorated with `@capture_logging` will fail if there are any tracebacks logged (using `write_traceback` or `writeFailure`) on the theory that these are unexpected errors indicating a bug. If you expected a particular traceback to be logged you can call `MemoryLogger.flush_tracebacks`, after which it will no longer cause a test failure. The result will be a list of traceback message dictionaries for the particular exception.

```
from unittest import TestCase
from eliot.testing import capture_logging

class MyTests(TestCase):
    def assertMythingBadPathLogging(self, logger):
        messages = logger.flush_tracebacks(OSError)
        self.assertEqual(len(messages), 1)

    @capture_logging(assertMythingBadPathLogging)
    def test_mythingBadPath(self, logger):
         mything = MyThing()
         # Trigger an error that will cause a OSError traceback to be logged:
         self.assertFalse(mything.load("/nonexistent/path"))
```

Testing Message and Action Structure

Eliot provides utilities for making assertions about the structure of individual messages and actions. The simplest method is using the `assertHasMessage` utility function which asserts that a message of a given `MessageType` has the given fields:

```
from eliot.testing import assertHasMessage, capture_logging

class LoggingTests(TestCase):
    @capture_logging(assertHasMessage, LOG_USER_REGISTRATION,
                     {u"username": u"john",
                      u"password": u"password",
                      u"age": 12})
    def test_registration(self, logger):
        """
        Registration adds entries to the in-memory database.
        """
        registry = UserRegistration()
        registry.register(u"john", u"password", 12)
        self.assertEqual(registry.db[u"john"], (u"passsword", 12))
```

`assertHasMessage` returns the found message and can therefore be used within more complex assertions. `assertHasAction` provides similar functionality for actions (see example below).

More generally, `eliot.testing.LoggedAction` and `eliot.testing.LoggedMessage` are utility classes to aid such testing. `LoggedMessage.of_type` lets you find all messages of a specific `MessageType`. A `LoggedMessage` has an attribute `message` which contains the logged message dictionary. For example, we could rewrite the registration logging test above like so:

```
from eliot.testing import LoggedMessage, capture_logging

class LoggingTests(TestCase):
    def assertRegistrationLogging(self, logger):
        """
        Logging assertions for test_registration.
        """
        logged = LoggedMessage.of_type(logger.messages, LOG_USER_REGISTRATION)[0]
        assertContainsFields(self, logged.message,
                             {u"username": u"john",
                              u"password": u"password",
                              u"age": 12}))
```

```
    @capture_logging(assertRegistrationLogging)
    def test_registration(self, logger):
        """
        Registration adds entries to the in-memory database.
        """
        registry = UserRegistration()
        registry.register(u"john", u"password", 12)
        self.assertEqual(registry.db[u"john"], (u"passsword", 12))
```

Similarly, `LoggedAction.of_type` finds all logged actions of a specific `ActionType`. A `LoggedAction` instance has `start_message` and `end_message` containing the respective message dictionaries, and a `children` attribute containing a list of child `LoggedAction` and `LoggedMessage`. That is, a `LoggedAction` knows about the messages logged within its context. `LoggedAction` also has a utility method `descendants()` that returns an iterable of all its descendants. We can thus assert that a particular message (or action) was logged within the context of another action.

For example, let's say we have some code like this:

```
LOG_SEARCH = ActionType(...)
LOG_CHECK = MessageType(...)

class Search:
    def search(self, servers, database, key):
        with LOG_SEARCH(database=database, key=key):
        for server in servers:
            LOG_CHECK(server=server).write()
            if server.check(database, key):
                return True
        return False
```

We want to assert that the LOG_CHECK message was written in the context of the LOG_SEARCH action. The test would look like this:

```
from eliot.testing import LoggedAction, LoggedMessage, capture_logging
import searcher

class LoggingTests(TestCase):
    @capture_logging(None)
    def test_logging(self, logger):
        searcher = Search()
        servers = [buildServer(), buildServer()]

        searcher.search(servers, "users", "theuser")
        action = LoggedAction.of_type(logger.messages, searcher.LOG_SEARCH)[0]
        messages = LoggedMessage.of_type(logger.messages, searcher.LOG_CHECK)
        # The action start message had the appropriate fields:
        assertContainsFields(self, action.start_message,
                             {"database": "users", "key": "theuser"})
        # Messages were logged in the context of the action
        self.assertEqual(action.children, messages)
        # Each message had the respective server set.
        self.assertEqual(servers, [msg.message["server"] for msg in messages])
```

Or we can simplify further by using `assertHasMessage` and `assertHasAction`:

```
from eliot.testing import LoggedAction, LoggedMessage, capture_logging
import searcher
```

```
class LoggingTests(TestCase):
    @capture_logging(None)
    def test_logging(self, logger):
        searcher = Search()
        servers = [buildServer(), buildServer()]

        searcher.search(servers, "users", "theuser")
        action = assertHasAction(self, logger, searcher.LOG_SEARCH, succeeded=True,
                                 startFields={"database": "users",
                                              "key": "theuser"})

        # Messages were logged in the context of the action
        messages = LoggedMessage.of_type(logger.messages, searcher.LOG_CHECK)
        self.assertEqual(action.children, messages)
        # Each message had the respective server set.
        self.assertEqual(servers, [msg.message["server"] for msg in messages])
```

Restricting Testing to Specific Messages

If you want to only look at certain messages when testing you can log to a specific `eliot.Logger` object. The messages will still be logged normally but you will be able to limit tests to only looking at those messages.

You can log messages to a specific `Logger`:

```
from eliot import Message, Logger

class YourClass(object):
    logger = Logger()

    def run(self):
        # Create a message with two fields, "key" and "value":
        msg = Message.new(key=123, value=u"hello")
        # Write the message:
        msg.write(self.logger)
```

As well as actions:

```
from eliot import start_action

logger = Logger()

with start_action(logger, action_type=u"store_data"):
    x = get_data()
    store_data(x)
```

Or actions created from `ActionType`:

```
from eliot import Logger

  from myapp.logtypes import LOG_USER_REGISTRATION

  class UserRegistration(object):

      logger = Logger()

      def __init__(self):
          self.db = {}
```

```
    def register(self, username, password, age):
        self.db[username] = (password, age)
        msg = LOG_USER_REGISTRATION(
             username=username, password=password, age=age)
        # Notice use of specific logger:
        msg.write(self.logger)
```

The tests would then need to do two things:

1. Decorate your test with `validate_logging` instead of `capture_logging`.
2. Override the logger used by the logging code to use the one passed in to the test.

For example:

```
from eliot.testing import LoggedMessage, validate_logging

class LoggingTests(TestCase):
    def assertRegistrationLogging(self, logger):
        """
        Logging assertions for test_registration.
        """
        logged = LoggedMessage.of_type(logger.messages, LOG_USER_REGISTRATION)[0]
        assertContainsFields(self, logged.message,
                             {u"username": u"john",
                              u"password": u"password",
                              u"age": 12}))

    # validate_logging only captures log messages logged to the MemoryLogger
    # instance it passes to the test:
    @validate_logging(assertRegistrationLogging)
    def test_registration(self, logger):
        """
        Registration adds entries to the in-memory database.
        """
        registry = UserRegistration()
        # Override logger with one used by test:
        registry.logger = logger
        registry.register(u"john", u"password", 12)
        self.assertEqual(registry.db[u"john"], (u"password", 12))
```

Asyncio Coroutine Support

If you're using `asyncio` coroutines in Python 3.5 or later (`async def yourcoro()` and `await yourcoro()`) together with Eliot, you need to run the following before doing any logging:

```
import eliot
eliot.use_asyncio_context()
```

Why you need to do this

By default Eliot provides a different "context" for each thread. That is how `with start_action(action_type='my_action'):` works: it records the current action on this context.

When using coroutines you end up with the same context being used with different coroutines, since they share the same thread. Calling `eliot.use_asyncio_context()` makes sure each coroutine gets its own context, so `with start_action()` in one coroutine doesn't interfere with another.

However, Eliot will do the right thing for nested coroutines. Specifically, coroutines called via `await a_coroutine()` will inherit the logging context from the calling coruotine.

Limitations

- I haven't tested the Python 3.4 `yield from` variation.
- This doesn't support other event loops (Curio, Trio, Tornado, etc.). If you want these supported please file an issue: https://github.com/ScatterHQ/eliot/issues/new There is talk of adding the concept of a coroutine context to Python 3.7 or perhaps 3.8, in which case it will be easier to automatically support all frameworks.

Using Eliot with Twisted

Eliot provides a variety of APIs to support integration with the Twisted networking framework.

Non-blocking Destinations

`eliot.logwriter.ThreadedWriter` is a logging destination that wraps a blocking destination and writes to it in a non-reactor thread. This is useful because it keeps the Twisted reactor from blocking, e.g. if you're writing to a log file and the hard drive is overloaded. `ThreadedWriter` is a Twisted `Service` and starting it will call `add_destinations` for you and stopping it will call `remove_destination`; there is no need to call those directly.

```
"""
Output an Eliot message to a log file using the threaded log writer.
"""
from __future__ import unicode_literals, print_function

from twisted.internet.task import react

from eliot.logwriter import ThreadedWriter
from eliot import Message, FileDestination


def main(reactor):
    print("Logging to example-eliot.log...")
    logWriter = ThreadedWriter(
        FileDestination(file=open("example-eliot.log", "ab")), reactor)

    # Manually start the service, which will add it as a
    # destination. Normally we'd register ThreadedWriter with the usual
    # Twisted Service/Application infrastructure.
    logWriter.startService()

    # Log a message:
    Message.log(value="hello", another=1)

    # Manually stop the service.
    done = logWriter.stopService()
    return done
```

```
if __name__ == '__main__':
    react(main, [])
```

If you want log rotation you can pass in an `eliot.FileDestination` wrapping one of the classes from twisted.python.logfile as the destination file.

`twisted.logger` integration

If you wish you can direct Eliot logs to Twisted's logging subsystem, if that is the primary logging system you're using.

```
from eliot import add_destinations
from eliot.twisted import TwistedDestination

add_destinations(TwistedDestination())
```

Trial Integration

If you're using Twisted's `trial` program to run your tests you can redirect your Eliot logs to Twisted's logs by calling `eliot.twisted.redirectLogsForTrial()`. This function will automatically detect whether or not it is running under `trial`. If it is then you will be able to read your Eliot logs in `_trial_temp/test.log`, where `trial` writes out logs by default. If it is not running under `trial` it will not do anything. In addition calling it multiple times has the same effect as calling it once.

The way you use it is by putting it in your package's `__init__.py`: it will do the right thing and only redirect if you're using `trial`. Take care if you are separately redirecting Twisted logs to Eliot; you should make sure not to call `redirectLogsForTrial` in that case so as to prevent infinite loops.

Logging Failures

`eliot.writeFailure` is the equivalent of `eliot.write_traceback`, only for `Failure` instances:

```
from eliot import writeFailure

class YourClass(object):

    def run(self):
        d = dosomething()
        d.addErrback(writeFailure)
```

Actions and Deferreds

An additional set of APIs is available to help log actions when using Deferreds. To understand why, consider the following example:

```
from eliot import start_action

def go():
    action = start_action(action_type=u"yourapp:subsystem:frob")
    with action:
        d = Deferred()
```

```
        d.addCallback(gotResult, x=1)
        return d
```

This has two problems. First, `gotResult` is not going to run in the context of the action. Second, the action finishes once the `with` block finishes, i.e. before `gotResult` runs. If we want `gotResult` to be run in the context of the action and to delay the action finish we need to do some extra work, and manually wrapping all callbacks would be tedious.

To solve this problem you can use the `eliot.twisted.DeferredContext` class. It grabs the action context when it is first created and provides the same API as `Deferred` (`addCallbacks` and friends), with the difference that added callbacks run in the context of the action. When all callbacks have been added you can indicate that the action should finish after those callbacks have run by calling `DeferredContext.addActionFinish`. As you would expect, if the `Deferred` fires with a regular result that will result in success message. If the `Deferred` fires with an errback that will result in failure message. Finally, you can unwrap the `DeferredContext` and access the wrapped `Deferred` by accessing its `result` attribute.

```
from eliot import start_action
from eliot.twisted import DeferredContext

def go():
    with start_action(action_type=u"your_type").context() as action:
        d = DeferredContext(Deferred())
        # gotResult(result, x=1) will be called in the context of the action:
        d.addCallback(gotResult, x=1)
        # After gotResult finishes, finish the action:
        d.addActionFinish()
        # Return the underlying Deferred:
        return d.result
```

Outputting Logs

Configuring Logging Output

You can register "destinations" to handle logging output. A destination is a callable that takes a message dictionary. For example, if we want each message to be encoded in JSON and written on a new line on stdout:

```
import json, sys
from eliot import add_destinations

def stdout(message):
    sys.stdout.write(json.dumps(message) + "\n")
add_destinations(stdout)
```

Up to a 1000 messages will be buffered in memory until the first set of destinations are added, at which point those messages will be delivered to newly added set of destinations.

Outputting to Files

You can create a destination that logs to a file by calling `eliot.FileDestination(file=yourfile)`. Each Eliot message will be encoded in JSON and written on a new line. As a short hand you can call `eliot.to_file` which will create the destination and then add it. For example:

```
from eliot import to_file
to_file(open("eliot.log", "ab"))
```

Note: This destination is blocking: if writing to a file takes a long time your code will not be able to proceed until writing is done. If you're using Twisted you can wrap a `eliot.FileDestination` with a non-blocking *eliot.logwriter.ThreadedWriter*. This allows you to log to a file without blocking the Twisted `reactor`.

Adding Fields to All Messages

Sometimes you want to add a field to all messages output by your process, regardless of destination. For example if you're aggregating logs from multiple processes into a central location you might want to include a field `process_id` that records the name and process id of your process in every log message. Use the `eliot.add_global_fields` API to do so, e.g.:

```
import os, sys
from eliot import add_global_fields

add_global_fields(process_id="%s:%d" % (sys.argv[0], os.getpid()))
```

You should call `add_global_fields` before `add_destinations` to ensure all messages get the global fields.

Journald

`journald` is the native logging system on Linux operating systems that use `systemd` with support for structured, indexed log storage. Eliot provides native `journald` support, with the following features:

- The default message field (`MESSAGE`) stores the Eliot message as JSON.
- Failed actions get priority 3 ("err") and tracebacks get priority 2 ("crit").
- The `ELIOT_TASK` field stores the task UUID.
- The `ELIOT_TYPE` field stores the message or action type if available.
- The `SYSLOG_IDENTIFIER` stores `sys.argv[0]`.

Installation

Journald requires additional libraries that are not installed by default by Eliot. You can install them by running:

```
$ pip install eliot[journald]
```

Generating logs

The following example demonstrates how to enable `journald` output.

```
"""
Write some logs to journald.
"""

from __future__ import print_function

```

```
from eliot import Message, start_action, add_destinations
from eliot.journald import JournaldDestination

add_destinations(JournaldDestination())


def divide(a, b):
    with start_action(action_type="divide", a=a, b=b):
        return a / b

print(divide(10, 2))
Message.log(message_type="inbetween")
print(divide(10, 0))
```

Querying logs

The `journalctl` utility can be used to extract logs from `journald`. Useful options include `--all` which keeps long fields from being truncated and `--output cat` which only outputs the body of the `MESSAGE` field, i.e. the JSON-serialized Eliot message.

Let's generate some logs:

```
$ python journald.py
```

We can find all messages with a specific type:

```
$ sudo journalctl --all --output cat ELIOT_TYPE=inbetween | eliot-prettyprint
32ab1286-c356-439d-86f8-085fec3b65d0 -> /1
2015-09-23 21:26:37.972403Z
  message_type: inbetween
```

We can filter to those that indicate errors:

```
$ sudo journalctl --all --output cat --priority=err ELIOT_TYPE=divide | eliot-
↪prettyprint
ce64eb77-bb7f-4e69-83f8-07d7cdaffaca -> /2
2015-09-23 21:26:37.972945Z
  action_type: divide
  action_status: failed
  exception: exceptions.ZeroDivisionError
  reason: integer division or modulo by zero
```

We can also search by task UUID, in which case `eliot-tree` can also be used to process the output:

```
$ sudo journalctl --all --output cat ELIOT_TASK=ce64eb77-bb7f-4e69-83f8-07d7cdaffaca␣
↪| eliot-tree
ce64eb77-bb7f-4e69-83f8-07d7cdaffaca
+-- divide@1/started
    |-- a: 10
    |-- b: 0
    `-- timestamp: 2015-09-23 17:26:37.972716
    +-- divide@2/failed
        |-- exception: exceptions.ZeroDivisionError
        |-- reason: integer division or modulo by zero
        `-- timestamp: 2015-09-23 17:26:37.972945
```

Using Logstash and ElasticSearch to Process Eliot Logs

ElasticSearch is a search and analytics engine which can be used to store Eliot logging output. The logs can then be browsed by humans using the Kibana web UI, or on the command-line using the logstash-cli tool. Automated systems can access the logs using the ElasticSearch query API. Logstash is a log processing tool that can be used to load Eliot log files into ElasticSearch. The combination of ElasticSearch, Logstash, and Kibana is sometimes referred to as ELK.

Example Logstash Configuration

Assuming each Eliot message is written out as a JSON message on its own line (which is the case for `eliot.to_file()` and `eliot.logwriter.ThreadedFileWriter`), the following Logstash configuration will load these log messages into an in-process ElasticSearch database:

`logstash_standalone.conf`

```
input {
  stdin {
    codec => json_lines {
      charset => "UTF-8"
    }
  }
}

filter {
  date {
    # Parse Eliot timestamp filed into the special @timestamp field Logstash
    # expects:
    match => [ "timestamp", "UNIX" ]
    target => ["@timestamp"]
  }
}

output {
  # Stdout output for debugging:
  stdout {
    codec => rubydebug
  }

  elasticsearch {
    # Documents in ElasticSearch are identified by tuples of (index, mapping
    # type, document_id).
    # References:
    # - http://logstash.net/docs/1.3.2/outputs/elasticsearch
    # - http://stackoverflow.com/questions/15025876/what-is-an-index-in-elasticsearch

    # We make the document id unique (for a specific index/mapping type pair) by
    # using the relevant Eliot fields. This means replaying messages will not
    # result in duplicates, as long as the replayed messages end up in the same
    # index (see below).
    document_id => "%{task_uuid}_%{task_level}"

    # By default logstash sets the index to include the current date. When we
    # get to point of replaying log files on startup for crash recovery we might
    # want to use the last modified date of the file instead of current date,
    # otherwise we'll get documents ending up in wrong index.

    #index => "logstash-%{+YYYY.MM.dd}"
```

```
    index_type => "Eliot"

    # In a centralized ElasticSearch setup we'd be specifying host/port
    # or some such. In this setup we run it ourselves:
    embedded => true
  }
}
```

We can then pipe JSON messages from Eliot into ElasticSearch using Logstash:

```
$ python examples/stdout.py | logstash web -- agent --config logstash_standalone.conf
```

You can then use the Kibana UI to search and browse the logs by visiting http://localhost:9292/.

Reading Logs

Reading Eliot Logs

Eliot includes a command-line tool that makes it easier to read JSON-formatted Eliot messages:

```
$ python examples/stdout.py | eliot-prettyprint
af79ef5c-280c-4b9f-9652-e14deb85d52d@/1
2015-09-25T19:41:37.850208Z
  another: 1
  value: hello

0572701c-e791-48e8-9dd2-1fb3bf06826f@/1
2015-09-25T19:41:38.050767Z
  another: 2
  value: goodbye
```

The third-party eliot-tree tool renders JSON-formatted Eliot messages into a tree visualizing the tasks' actions. Unlike `eliot-prettyprint` it may not be able to format all messages if some of a task's messages are missing.

Message Fields in Depth

Built-in Fields

A number of fields are reserved by Eliot's built-in message structure and should not be added to messages you create.

All messages contain `task_uuid` and `task_level` fields. Each message is uniquely identified by the combined values in these fields. For more information see the *actions and tasks* documentation.

In addition, the following field will also be present:

- `timestamp`: Number of seconds since Unix epoch as a `float` (the output of `time.time()`). Since system time may move backwards and resolution may not be high enough this cannot be relied on for message ordering.

Assuming you are using `MessageType` and `ActionType` every logged message will have either `message_type` or `action_type` fields depending whether they originated as a standalone message or as the start or end of an action. Present in regular messages:

- `message_type`: The type of the message, e.g. `"yourapp:yoursubsystem:yourmessage"`.

Present in action messages:

- `action_type`: The type of the action, e.g. `"yourapp:yoursubsystem:youraction"`.
- `action_status`: One of `"started"`, `"succeeded"` or `"failed"`.

The following fields can be added to your messages, but should preserve the same meaning:

- `exception`: The fully qualified Python name (i.e. import path) of an exception type, e.g. `"yourpackage.yourmodule.YourException"`.
- `reason`: A prose string explaining why something happened. Avoid usage if possible, better to use structured data.
- `traceback`: A string with a traceback.
- `system`: A string, a subsystem in your application, e.g. `"yourapp:yoursubsystem"`.

User-Created Fields

It is recommended, but not necessary (and perhaps impossible across organizations) that fields with the same name have the same semantic content.

Contributing to Eliot

To run the full test suite, the Daemontools package should be installed.

All modules should have the `from __future__ import unicode_literals` statement, to ensure Unicode is used by default.

Coding standard is PEP8, with the only exception being camel case methods for the Twisted-related modules. Some camel case methods remain for backwards compatibility reasons with the old coding standard.

You should use `yapf` to format code.

CHAPTER 3

Project Information

Eliot is maintained by Itamar Turner-Trauring, and released under the Apache 2.0 License.