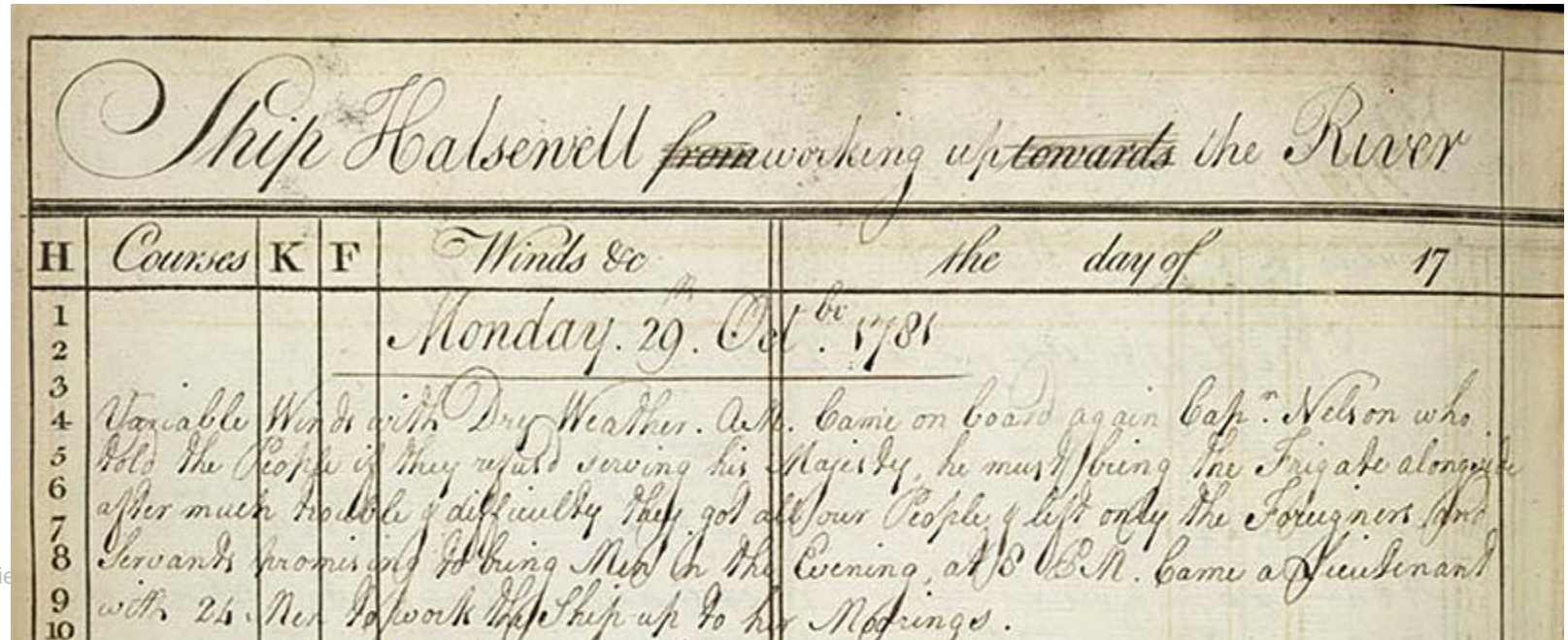
Why log?

Debugging
- what is going wrong and why?
- what happened? why is something missing/wrong/etc?

Short-term monitoring
- is my change running as expected?
- what changed after a release (number of events, elapsed time, etc)
- activity feed for engineering (and maybe ops?)

Long-term monitoring
- sound an alarm if code runs too often / not often enough / throws errors


Ship Halsewell ~~from~~ working up ~~towards~~ the River

H	Courses	K	F	Winds &c	the day of 17
1 2 3				Monday. 29. Oct.r 1781	
4 5 6 7 8 9 10	Variable Winds with Dry Weather. At 8 Came on board again Capt.n Nelson who told the People if they would serving his Majesty, he must bring the Frigate along side after much trouble & difficulty they got all our People & left only the Foreigners and Servants promising to bring Men in the Evening. at 10 P.M. Came a Lieutenant with 24 Men to work the Ship up to her Moorings.				

Datadog logging demo

Quick tour
- search and facets
- view in context, vs previous
- card: default fields, add column
- saved view

Debugging
- CPU: alert, show logs, analytics
- Keen errors: filter by time and error, analytics; vs previous

Short-term monitoring
- app activity feed
- Quicksight activity feed

Long-term monitoring
- log to metric to monitor

Existing logs
- transformations
- standard attributes
- rehydrate vs S3

Python logging: what does this do?

```
import logging

log = logging.getLogger(__name__)

>>> log.debug("debug message", extra={"elapsed": 0.5})
>>>

>>> log.info("info message", extra={"elapsed": 1.0})
>>>

>>> log.warn("warn message", extra={"elapsed": 1.5})
warn message

>>> log.error("error message", extra={"elapsed": 2.0})
error message

>>> try: 1/0 except: log.exception("exception message")
exception message
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Python logging

Default logging config:

- WARN+
- message only
- text format
- print traceback on exception

Configure logging to customize

- formatters - output structure and contents
- handlers - send (some) messages to a destination
- loggers - send a message

Python logging: formatters

A formatter describes how to create a string from a LogRecord. It can include any attributes from LogRecord.

- **levelname** is the log level (DEBUG, INFO, WARN, ERROR)
- **name** is the name of the logger (should be the module)
- **message** is the log message, with variables merged
- time (**asctime**) not included; we let Docker host add this
- also possibly interesting: **funcName**, **exc_info** (type, value, traceback)

A formatter can also use a custom class. Here, `pythonjsonlogger.jsonlogger.JsonFormatter` outputs the same fields in JSON format.

```
"formatters": {
    "standard": {
        "format": "%(levelname)s %(name)s %(message)s",
    },
    "json": {
        "class": "pythonjsonlogger.jsonlogger.JsonFormatter",
        "format": "%(levelname)s %(name)s %(message)s",
    }
},
```

Python logging: handlers

A handler tells the logging system which formatter to use to create the message, and how to output it.

We use `logging.StreamHandler` to output log messages to stderr.

- `test.py | tee test.log` goes to stderr
- `“stream”: “ext://sys.stdout”` to write to stdout

Python logging also has built-in support for various file logging strategies (file, rotating file, etc)

- We let Docker collect stderr from containers to files

```
    "handlers": {
        "console": {"class": "logging.StreamHandler",
"formatter": "standard"},
    },
```

Python logging: loggers

Loggers customize behavior based on the logger name.

- loggers inherit configuration from their parents (up the “root” logger)
- can change any part of the config
- can output to any number of handlers (for example, console and a file)

```
    "root": {"handlers": ["console"], "level":
"INFO",},
    "loggers": {
        "webapps": {"level": "DEBUG"},
        "sqlalchemy": {"level": "INFO"},
        "sqlalchemy.orm": {"level": "WARN"},
        "sqlalchemy.engine.base.Engine": {"level":
"WARN"}
    },
```

Python logging: dictionary config

```
LOGGING = {
    "formatters": {
        "standard": {
            "format": "%(levelname)s %(name)s %(message)s",
        },
        "json": {
            "class": "pythonjsonlogger.jsonlogger.JsonFormatter",
            "format": "%(levelname)s %(name)s %(message)s",
        }
    },
    "handlers": {
        "console": {"class": "logging.StreamHandler", "formatter":
"standard"},
    },
    "root": {"handlers": ["console"], "level": "INFO",},
    "loggers": {
        "webapps": {"level": "DEBUG"},
        # sqlalchemy is really noisy
        "sqlalchemy": {"level": "INFO"},
        "sqlalchemy.orm": {"level": "WARN"},
        "sqlalchemy.engine.base.Engine": {"level": "WARN"}
    },
}

logging.config.dictConfig(LOGGING)
log = logging.getLogger(__name__)  # use module path as the name
```

Python logging: configuration

Configure a logger directly

- add/update any part of config

```
 logging.getLogger().setLevel(logging.DEBUG)
 logging.getLogger().handlers[0].setFormatter(
   logging.Formatter(
     "%(asctime)s %(name)s %(levelname)-8s %
(message)s"
   )
)
```

Python logging: local (text)

```
>>>
logging.config.dictConfig(config.LOGGING_CONFIG["development
"])
>>> log = logging.getLogger("webapps.test")
>>> log.debug("debug message", extra={"elapsed": 0.5})
DEBUG webapps.test debug message
>>> log.info("info message", extra={"elapsed": 1.0})
INFO webapps.test info message
>>> log.warn("warn message", extra={"elapsed": 1.5})
WARNING webapps.test warn message
>>> log.error("warn message", extra={"elapsed": 2.0})
ERROR webapps.test warn message
>>> try: 1/0 except: log.exception("exception message",
extra={"elapsed": 2.5})
ERROR webapps.test exception message
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Python logging: JSON

Datadog recommends logging in JSON:

> “Logging in JSON is a best practice when centralizing your logs with a log management service, because machines can easily parse and analyze this standard, structured format. JSON format is also easily customizable to include any attributes you decide to add to each log format, so you won’t need to update your log processing pipelines every time you add or remove an attribute from your log format.”

This is especially useful for exceptions. With text output, a traceback is split over multiple lines, which makes it hard to search:

```
1 energy-api-2a docker_workerbg_1: [2020-08-31 10:01:01,194: ERROR/MainProcess] Task run-partial-bill-stitcher[88407414-3870-46c
OperationalError('(psycopg2.errors.QueryCanceled) canceling statement due to statement timeout\n',)
1 energy-api-2a docker_workerbg_1: Traceback (most recent call last):
1 energy-api-2a docker_workerbg_1:   File "/usr/local/lib/python3.6/site-packages/celery-3.1.9-py3.6.egg/celery/app/trace.py", l
1 energy-api-2a docker_workerbg_1:     R = retval = fun(*args, **kwargs)
1 energy-api-2a docker_workerbg_1:   File "/usr/local/lib/python3.6/site-packages/celery-3.1.9-py3.6.egg/celery/app/trace.py", l
1 energy-api-2a docker_workerbg_1:     return self.run(*args, **kwargs)
1 energy-api-2a docker_workerbg_1:   File "/app/webapps/lib/db.py", line 123, in _decorated
1 energy-api-2a docker_workerbg_1:     rval = fn(*args, **kwargs)
```

JSON makes it easy to add additional context to log messages:

- `log.info("info message", extra={"elapsed": 1.0})`

Python logging: Docker and JSON

Docker captures standard output and standard error of all containers

- default logging driver is json-file
- adds stream (stdout/stderr) and time
- one file per container

Production:

- `/var/lib/docker/containers/container_id/container_id-json.log`

Local:

- use `docker-compose logs -f api`
- with Docker desktop, these are stored inside the Docker VM:

```
docker run -it --rm --privileged --pid=host justincormack/
nsenter1
tail -n 5 /var/lib/docker/containers/0ed082*/0ed082*-json.log
{"log":"{\"levelname\": \"DEBUG\", \"name\":
\"webapps.views.root\", \"message\": \"api: log-performance in
0.019194s\"}
\n","stream":"stderr","time":"2020-08-31T19:01:45.8457994Z"}
```

Python logging: JSON configuration

Tell the console handler to use the json formatter:

```
    "formatters": {
        "json": {
            "class":
"pythonjsonlogger.jsonlogger.JsonFormatter",
            "format": "%(levelname)s %(name)s %
(message)s",
        }
    },
    "handlers": {
        "console": {"class": "logging.StreamHandler",
"formatter": "json"},
    },
```

Python logging: production (JSON)

```
>>> log = logging.getLogger("webapps.test")
>>> log.debug("debug message", extra={"elapsed": 0.5})
{"levelname": "DEBUG", "name": "webapps.test", "message":
"debug message", "elapsed": 0.5}
>>> log.info("info message", extra={"elapsed": 1.0})
{"levelname": "INFO", "name": "webapps.test", "message":
"info message", "elapsed": 1.0}
>>> log.warn("warn message", extra={"elapsed": 1.5})
{"levelname": "WARNING", "name": "webapps.test", "message":
"warn message", "elapsed": 1.5}
>>> log.error("warn message", extra={"elapsed": 2.0})
{"levelname": "ERROR", "name": "webapps.test", "message":
"warn message", "elapsed": 2.0}
>>> try: 1/0 except: log.exception("exception message",
extra={"elapsed": 2.5})
{"levelname": "ERROR", "name": "webapps.test", "message":
"exception message", "exc_info": "Traceback (most recent call
last):\n  File \"<stdin>\", line 2, in
<module>\nZeroDivisionError: division by zero", "elapsed":
2.5}
```

Python logging: pyramid exception logging

```
@view_config(context=Exception,
permission=pyramid_security.NO_PERMISSION_REQUIRED)
def handle_general_errors(error, request):
    try:
        extra = {"exception": type(error).__name__}
        req_keys = ["method", "url", "view_name",
"authenticated_userid", "domain"]
        for key in req_keys:
            extra[key] = getattr(request, key)
        if request.params:
            extra["params"] = request.params.mixed()
        log.exception(error, extra=extra)
    except Exception as e:
        traceback.print_exc()
```

Python logging: pyramid exception logging


```
{
    "levelname": "ERROR",
    "name": "webapps.views.errors",
    "message": "division by zero",
    "exc_info": "Traceback (most recent call last):\n  File \"/usr/
local/lib/python3.6/site-packages/pyramid-1.9.1-py3.6.egg/pyramid/
tweens.py\", line 39, in excview_tween\n    response =
handler(request)\n ... x = 1/0\nZeroDivisionError: division by zero",
    "exception": "ZeroDivisionError",
    "method": "POST",
    "url": "http://localhost/log-performance",
    "view_name": "log-performance",
    "authenticated_userid": "5e74d93b224d5f0c09409af9",
    "domain": "localhost",
    "params": {
        "account": "5e84d780224d5fede11decda",
        "user": "5e74d93b224d5f0c09409af9",
        "route": "admin.accounts.index",
        "elapsedTime": "0.4704050000002794",
        "url": "/admin/accounts"
    }
}
```

Why, Celery, why?

Our logs look like this (`%(levelname)s %(name)s %(message)s`)

huyangapi_1 | DEBUG gridium.huyangapi.views.auth user from Bearer=100

But celery logs look like this:

```
huyangapiworker_1 | [2020-09-08 16:41:26,968: INFO/Worker-1] active_timers_email start
```

Celery hijacks the loggers and uses its own config.
Use celery signals to copy our logging config back:

- `@celery.signals.after_setup_task_logger.connect`
  - `get_task_logger(__name__)`
  - includes task_id and task_name
- `@celery.signals.after_setup_logger.connect`

Logging message construction

Easy to add and (potentially) high value

- mini-design problem that's worth thinking about: how can you use it?
- write for humans and machines

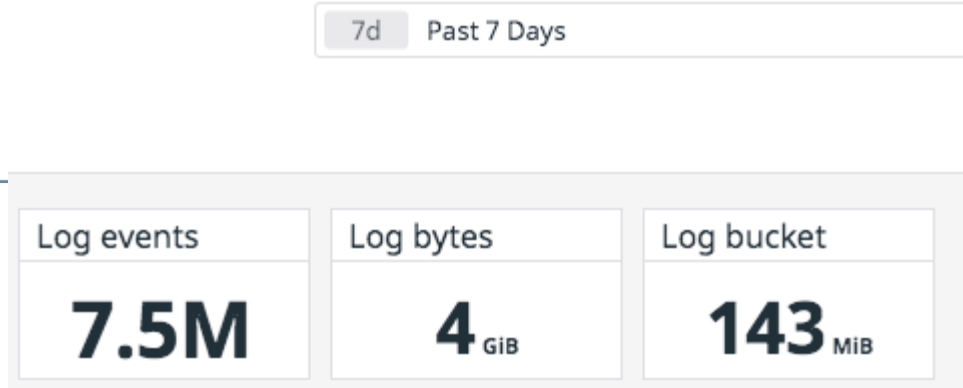
```
log.info(
    message  # full text searchable, human readable
    extra={"key": "value", ..}   # roll up, count, measure, monitor
)
```

Already included: `host, service, container, source, log_group, level, module`

With tasks (`get_task_logger`): `task_id, task_name, elapsed`

With log.exception: `exc_info` (traceback)

Switch to Datadog



Current 30 day free trial with all but analytics

analytics.log = 27% bytes, 1.6G/7d, 7G/mo

Datadog	**Papertrail**
$2.04/M log events	$230 for 25 GB, $150 for 16GB
15 day retention, S3 archive (+ $0.019/GB)	2 weeks search, 1 year archive
pay again to re-hydrate or get from S3	get from papertrail-cli or S3
$67/month (-analytics.log)	$230/month (+analytics.log) $150/month (-analytics.log)

Ready to switch?

More info

Datadog

Log explorer - start here

Generate metrics from logs

Processing logs to reformat and extract info from existing logs

Naming conventions and standard attributes

Python

Logging cookbook

Customizing celery logging

Celery task logging

Docker

Docker logging: a complete guide