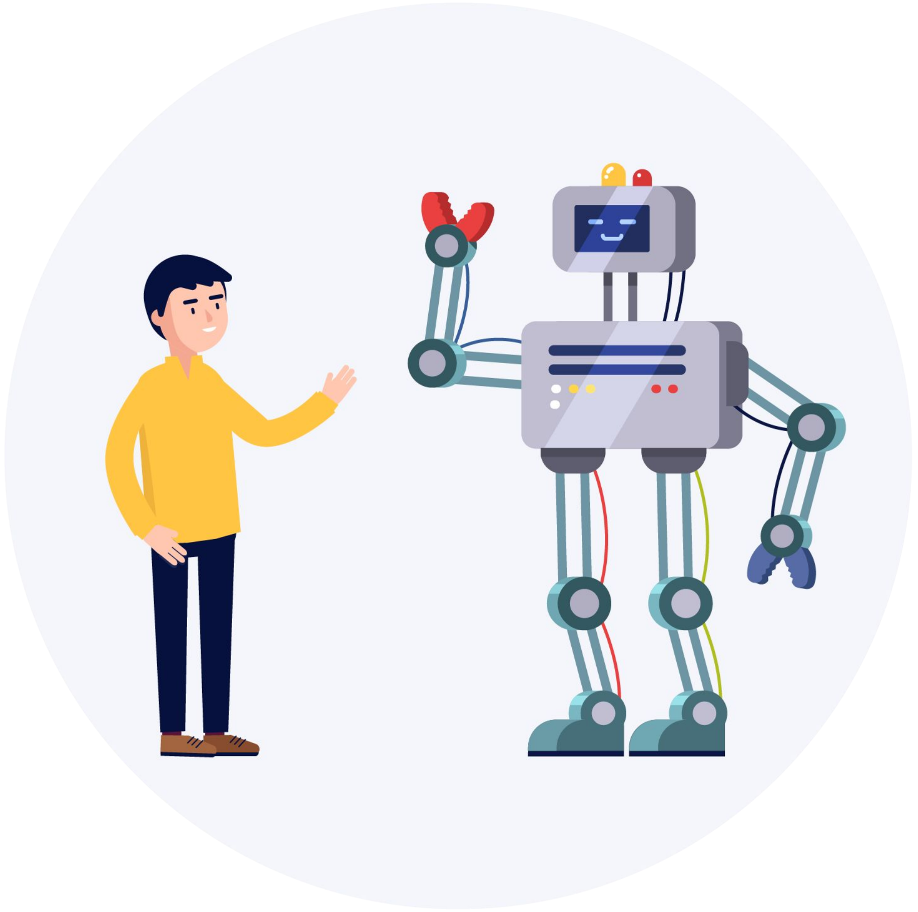
Logging in the cloud: Machines first, humans come second

Itiel Shwartz



About me

Joined **Rookout** as a first developer and production engineer

Previously Worked at **Forter** and **eBay** as a backend engineer

@itielshwartz on both Github and Twitter

personal blog at: https://etlsh.com

To log or not log?

Do we need logs in our program?

I write the code, I write the tests, and everything looks great!

Why the hell do I need to spend more time writing some bunch of lines nobody is going to read?

Prepare for the worst hope for the best

This is the mindset I want all of us to be in during this talk:

- **Your code is not as bulletproof as you might think**
- **Your code is not yours - someone else is going to debug it**
- **The infrastructure is going to fail you.**
- **The customer is ALWAYS going to disappoint you**

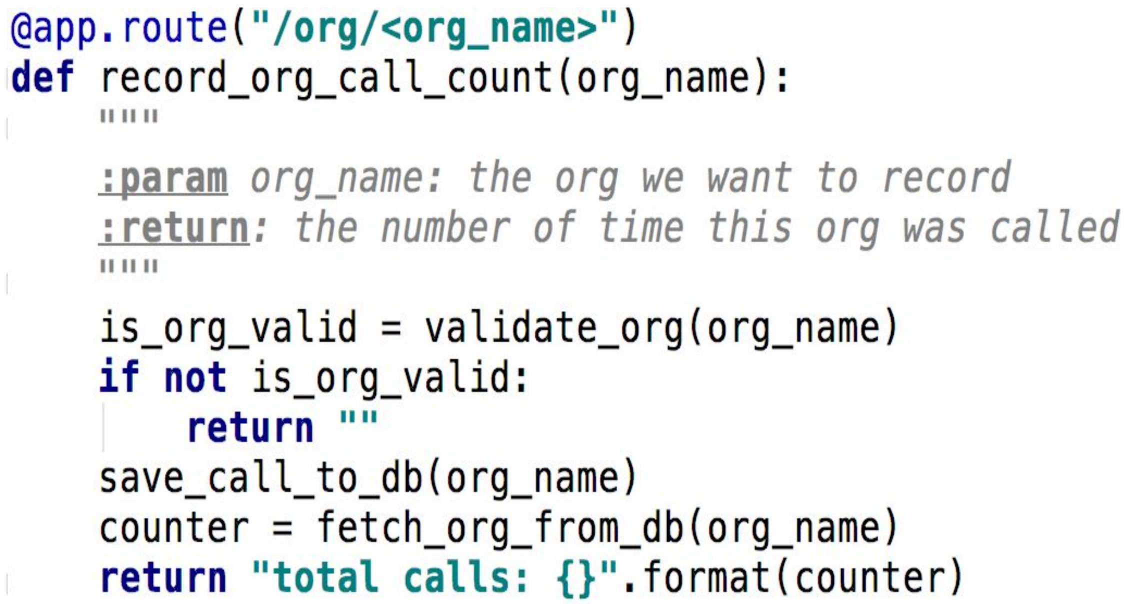
Logging journey

Our logging journey for today

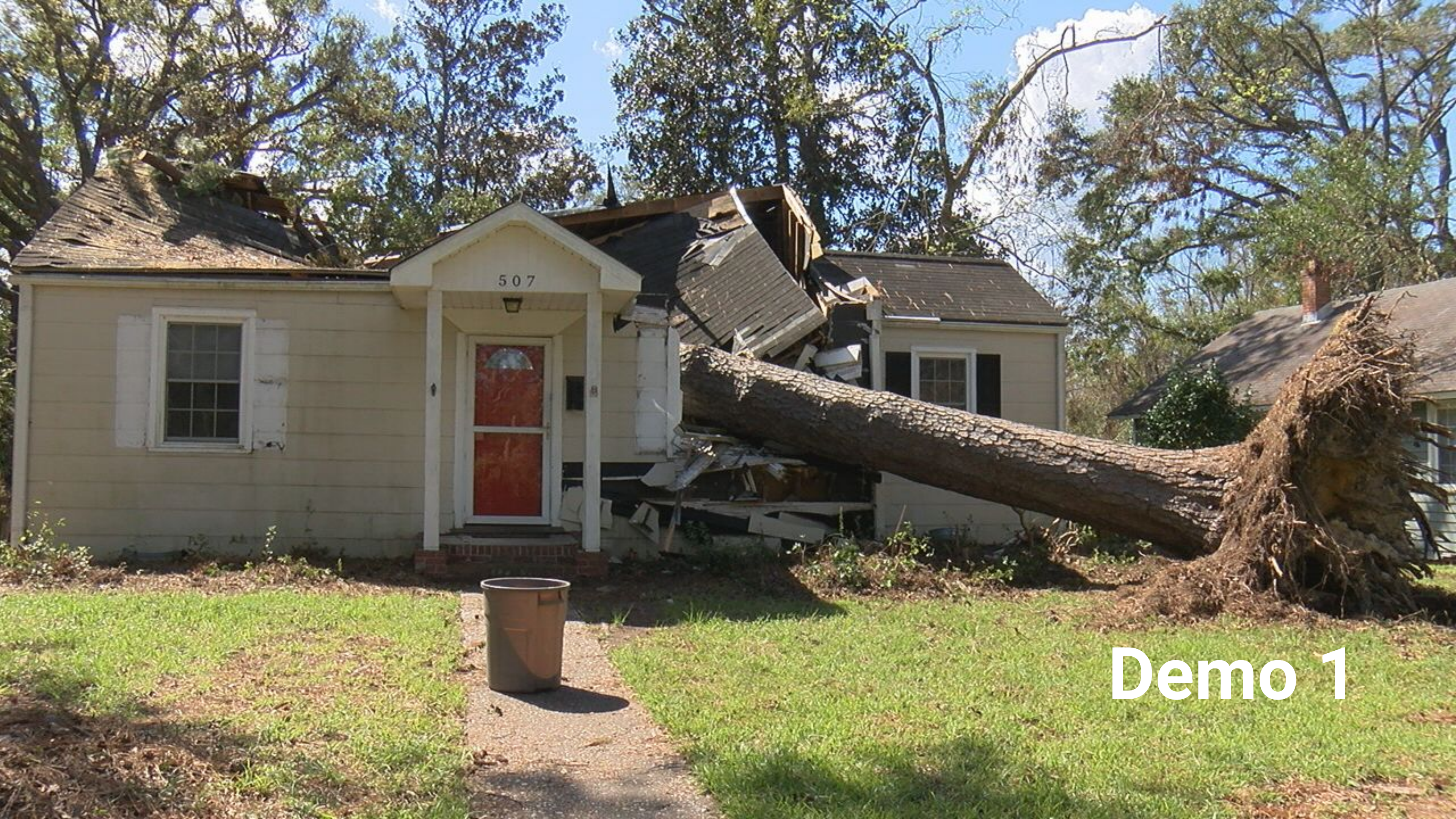
- We will start with a very small program.
- Then we will add some very basic logging capabilities
- As the talk continues we will add more and more logs and abilities to our program.
- No logs -> to print -> to logger -> to smarter logger -> to distribute tracing -> To the future of logging

No logs :(

- Get org name.
- Save it to DB.
- Return number of time it was called.
- Let’s try and run it



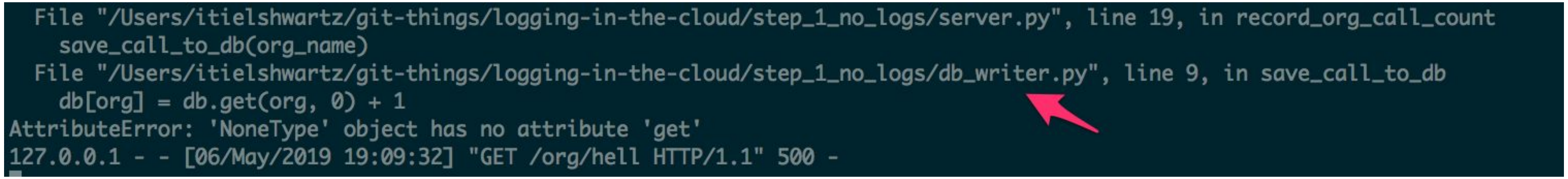
```
@app.route("/org/<org_name>")
def record_org_call_count(org_name):
    """
    :param org_name: the org we want to record
    :return: the number of time this org was called
    """
    is_org_valid = validate_org(org_name)
    if not is_org_valid:
        return ""
    save_call_to_db(org_name)
    counter = fetch_org_from_db(org_name)
    return "total calls: {}".format(counter)
```



Demo 1

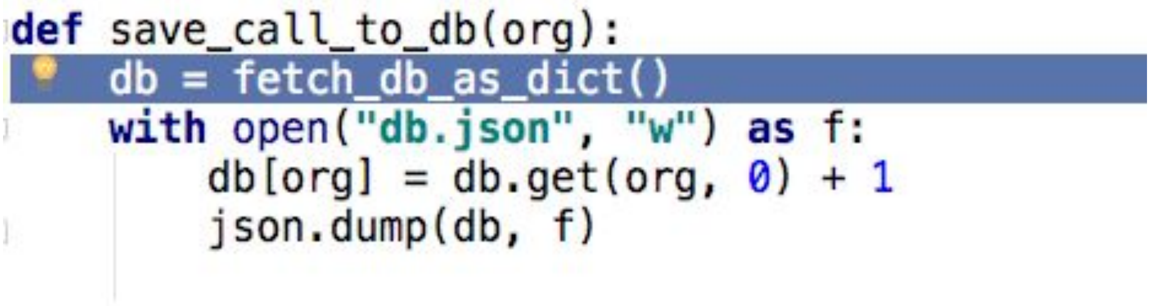
We have found our first BUG

The exception



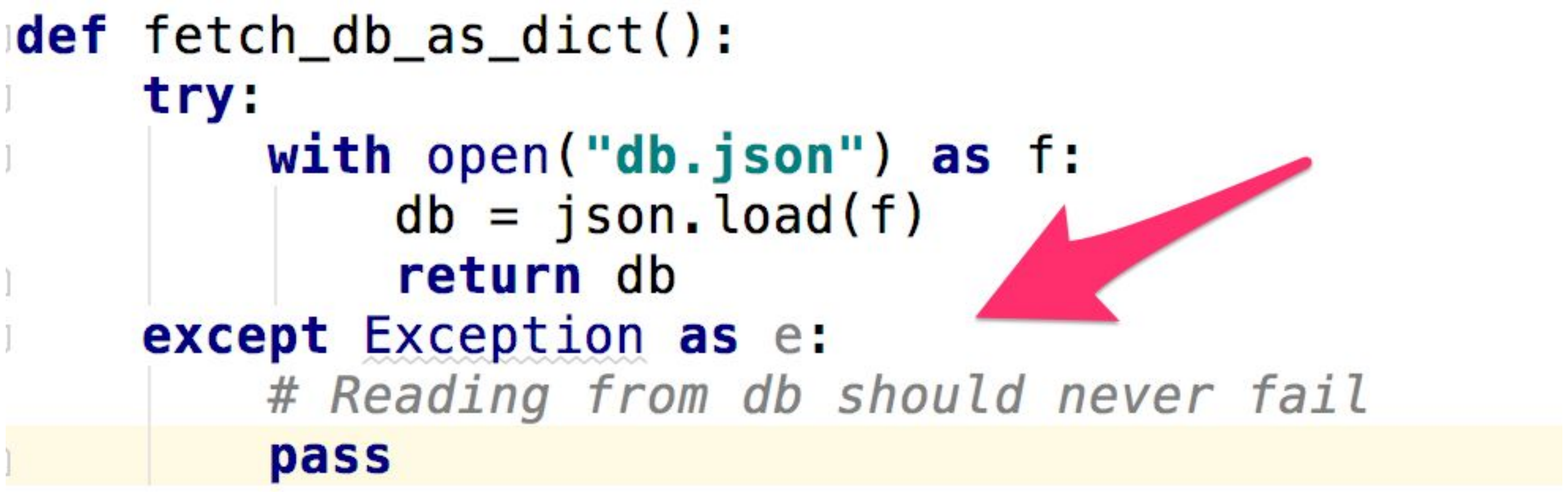
```
  File "/Users/itielshwartz/git-things/logging-in-the-cloud/step_1_no_logs/server.py", line 19, in record_org_call_count
    save_call_to_db(org_name)
  File "/Users/itielshwartz/git-things/logging-in-the-cloud/step_1_no_logs/db_writer.py", line 9, in save_call_to_db
    db[org] = db.get(org, 0) + 1
AttributeError: 'NoneType' object has no attribute 'get'
127.0.0.1 - - [06/May/2019 19:09:32] "GET /org/hell HTTP/1.1" 500 -
```

The code



```python
def save_call_to_db(org):
    db = fetch_db_as_dict()
    with open("db.json", "w") as f:
        db[org] = db.get(org, 0) + 1
        json.dump(db, f)
```

The real reason



```
def fetch_db_as_dict():
    try:
        with open("db.json") as f:
            db = json.load(f)
            return db
    except Exception as e:
        # Reading from db should never fail
        pass
```

Log the “unexpected”

- Remember: things will FAIL .
- **When you fail silently - you make it harder for the next guy**
- If you are not sure if this code will ever run -> log anyway :)

Log -> don’t print!

- The log record contains readily available diagnostic information such as the file name, full path, function, and line number of the logging event.
- Events logged in included modules are automatically accessible via the root logger to your application’s logging stream.
- Logging can be selectively silenced by log level

https://docs.python-guide.org/writing/logging/#or-print

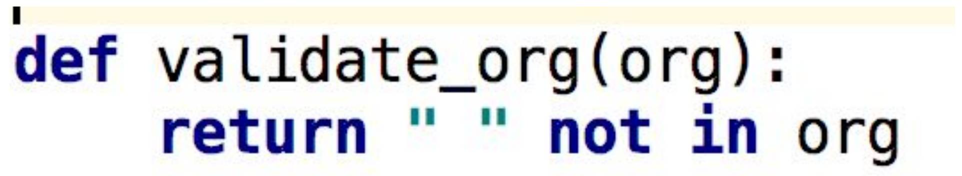
Demo 2

I know it doesn't work, don’t you?

- It’s the middle of the night.
- Suddenly a customer complains he can’t access the system
- You try it for yourself
- And fail
- No logs are in sight!

The “expected” unhappy flow

Something that someone “anticipated” has happened:



```python
def validate_org(org):
    return " " not in org
```

Sadly when this happens to our customer, at real time,

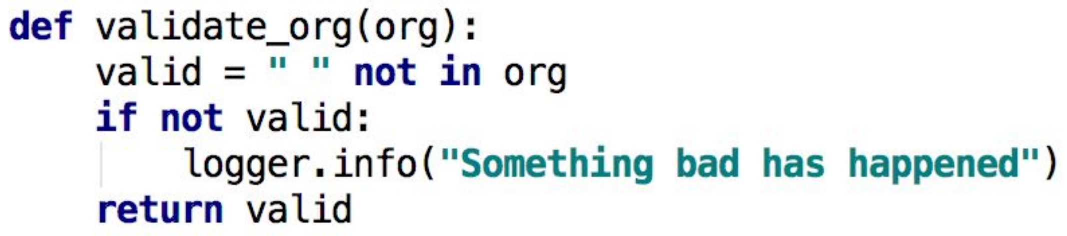
it’s almost impossible to find out.

Adding logs for human - first try

- A log that something bad has happened.

- The most “common” logs (mostly written while the feature was developed)

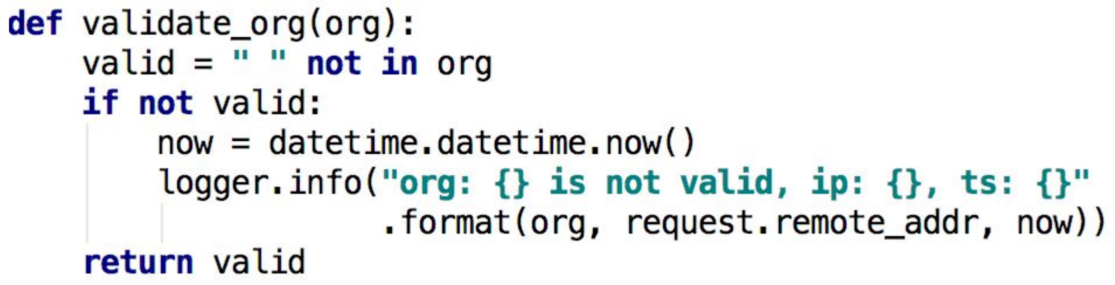
- What happened?
  To whom? When? why?



```python
def validate_org(org):
    valid = " " not in org
    if not valid:
        logger.info("Something bad has happened")
    return valid
```

Adding logs for human - second try

- This log has a lot more relevant data.
- It is very robust (maybe too much?)
- Format is very human oriented
- Is there some anti pattern here?



```python
def validate_org(org):
    valid = " " not in org
    if not valid:
        now = datetime.datetime.now()
        logger.info("org: {} is not valid, ip: {}, ts: {}"
                    .format(org, request.remote_addr, now))
    return valid
```

Logging for humans - recap

- **When writing logs we should not think about yourself, but the dev after you.**

- When unexpected things happen - you should log them.

- Each log should contain as much relevant data as possible.

- It’s OK to add and remove (or add) logs

Rise of the machines

Logging for Machines - JSON

- In most cases are data is going to arrive to some sort of centralized logging. (ELK/ Logz/ Splunk etc)

- We want to be able to do smart search, visualizations, alerts and more.

- When writing a “human-readable” log - we miss all of those features.

Logging for Machines - Example

Human: “org: **bla** is not valid, this is the time: TS"

-------- VS ------

Computer: {msg:”Org is not valid”, org:”bla”, time:ts}

A world of possibilities is now available:

- Visualization
- Alerts
- Aggregation with other logs
- Much more

Logging using machines (for machines)

There is a lot of shared-metadata available:

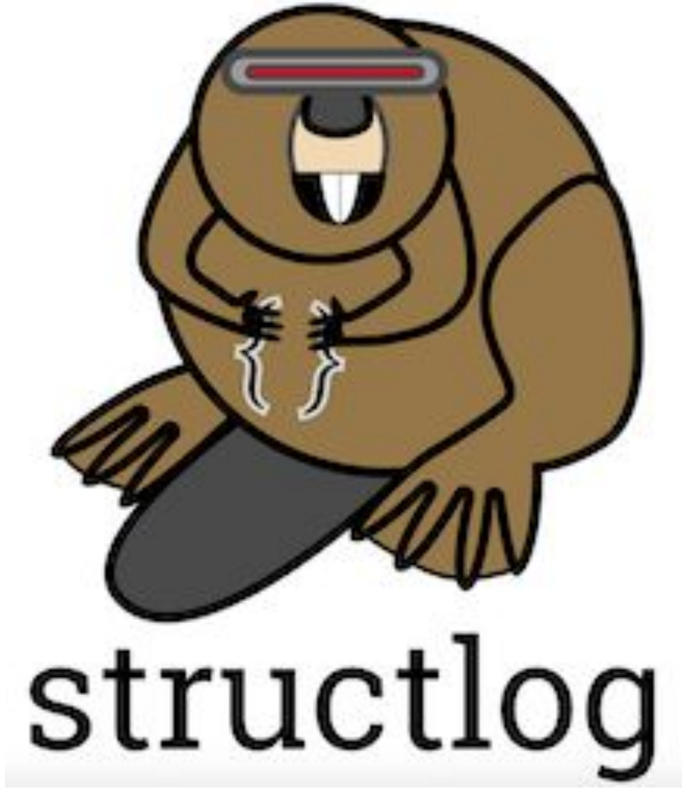
- Per request - org, ip, user etc
- Per env - commit hash, github, machine name
- By using a smarter logger we can get better logs - “for free”

Structlogs (Python)

structlog makes logging in Python less painful and more powerful by adding structure to your log entries.

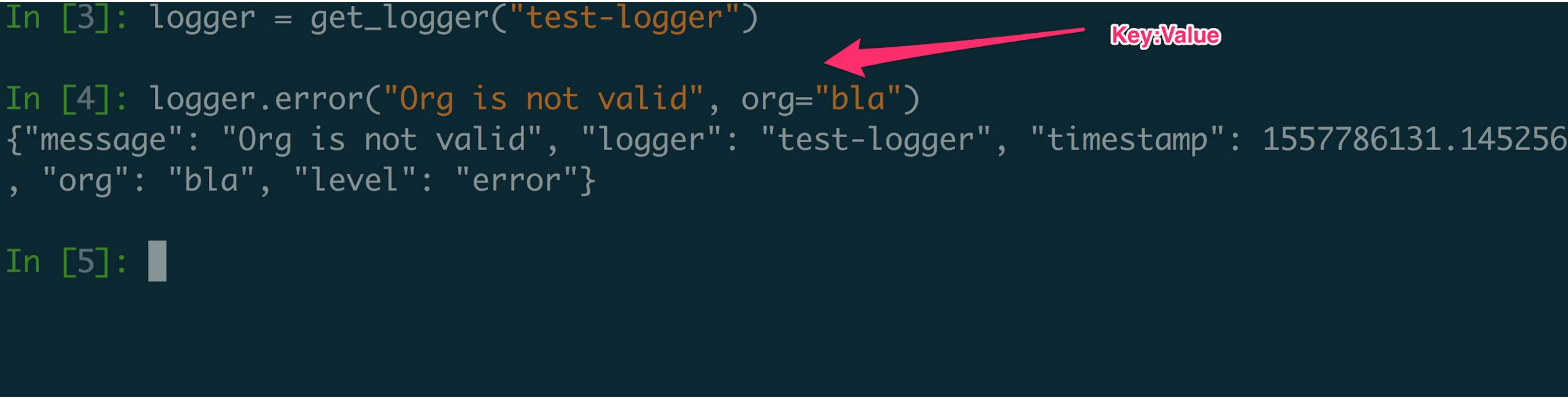
Important thing to remember:

Similar logging infra exist in any language



Easy to implement and use

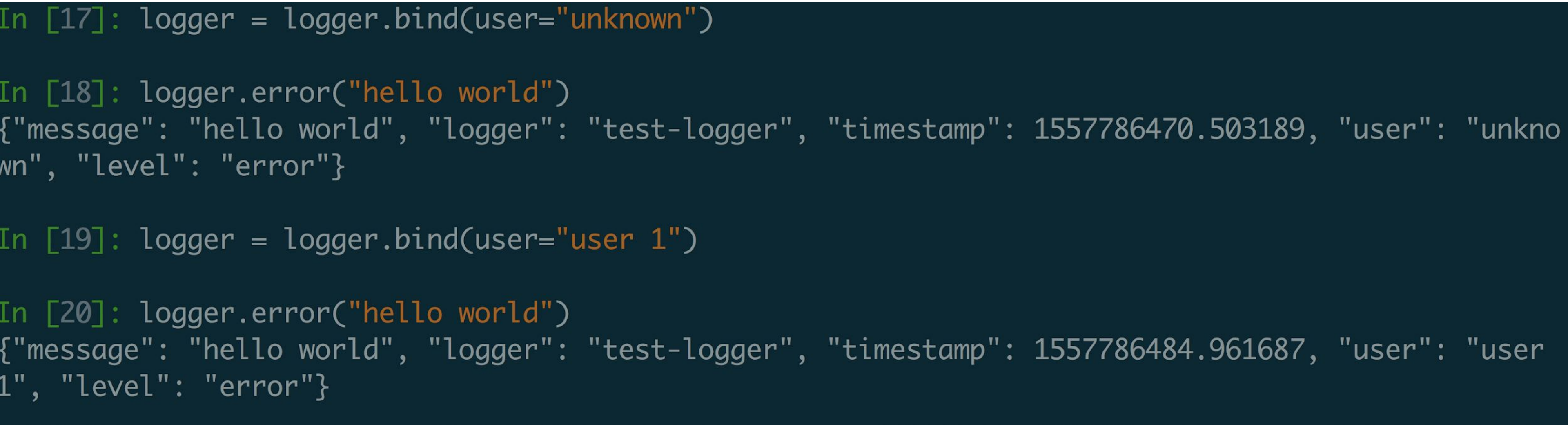
Using almost the same interface as a “normal” logger, while adding more capabilities on top:



```
In [3]: logger = get_logger("test-logger")

In [4]: logger.error("Org is not valid", org="bla")
{"message": "Org is not valid", "logger": "test-logger", "timestamp": 1557786131.145256
, "org": "bla", "level": "error"}

In [5]: 
```

Data binding

Since log entries are dictionaries, you can start binding and re-binding key/value pairs to your loggers to ensure they are present in every following logging call:

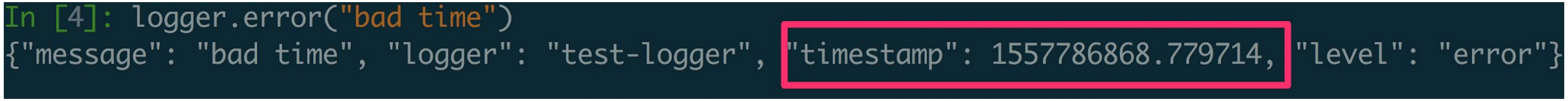


```
In [17]: logger = logger.bind(user="unknown")

In [18]: logger.error("hello world")
{"message": "hello world", "logger": "test-logger", "timestamp": 1557786470.503189, "user": "unkno
wn", "level": "error"}

In [19]: logger = logger.bind(user="user 1")

In [20]: logger.error("hello world")
{"message": "hello world", "logger": "test-logger", "timestamp": 1557786484.961687, "user": "user
1", "level": "error"}
```

Powerful Pipelines

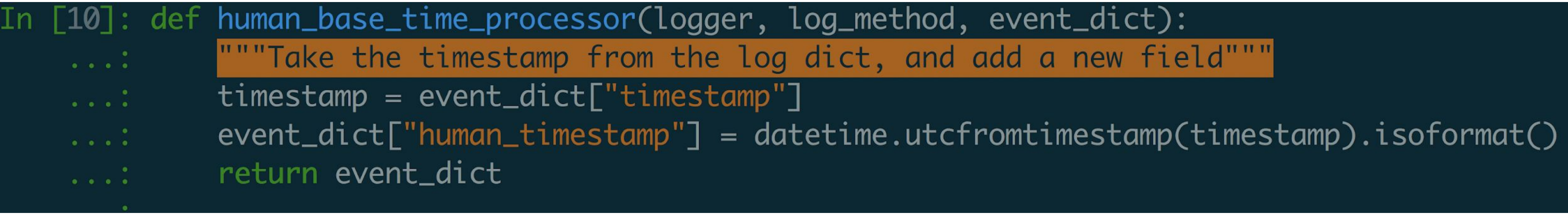
Each log entry goes through a processor pipeline that is just a chain of functions that receive a dictionary and return a new dictionary that gets fed into the next function. That allows for simple but powerful data manipulation.

Old log:



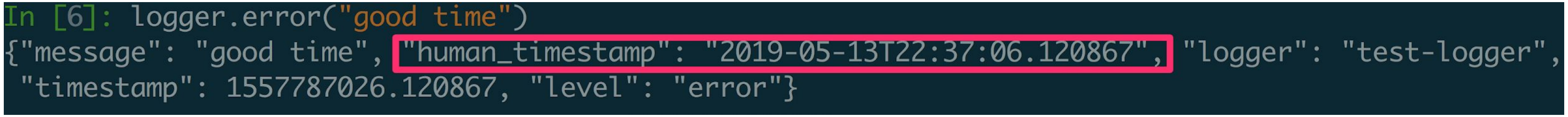
```
In [4]: logger.error("bad time")
{"message": "bad time", "logger": "test-logger", "timestamp": 1557786868.779714, "level": "error"}
```

A log enrichment function (happens once per log):



```
In [10]: def human_base_time_processor(logger, log_method, event_dict):
    ...:     """Take the timestamp from the log dict, and add a new field"""
    ...:     timestamp = event_dict["timestamp"]
    ...:     event_dict["human_timestamp"] = datetime.utcfromtimestamp(timestamp).isoformat()
    ...:     return event_dict
    ...:
```

New log:

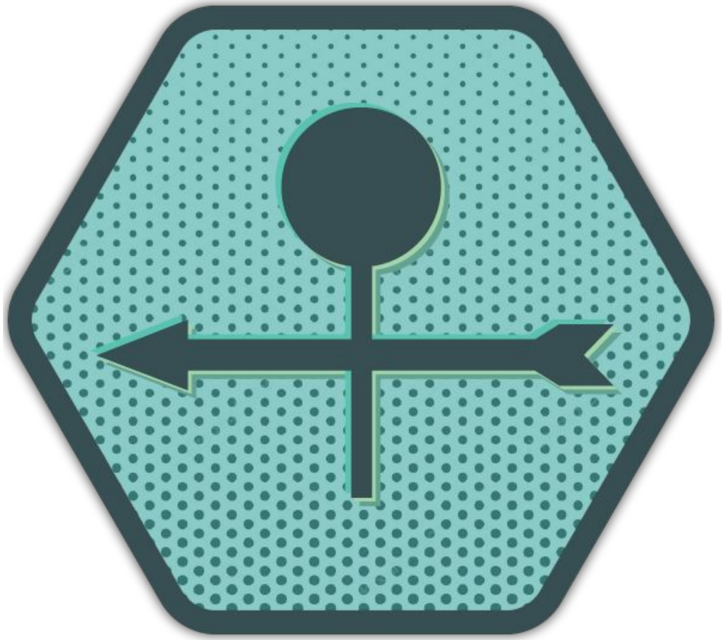


```
In [6]: logger.error("good time")
{"message": "good time", "human_timestamp": "2019-05-13T22:37:06.120867", "logger": "test-logger",
 "timestamp": 1557787026.120867, "level": "error"}
```

Logs :) logs :-0 logs :(

TOO MUCH LOGS!!!

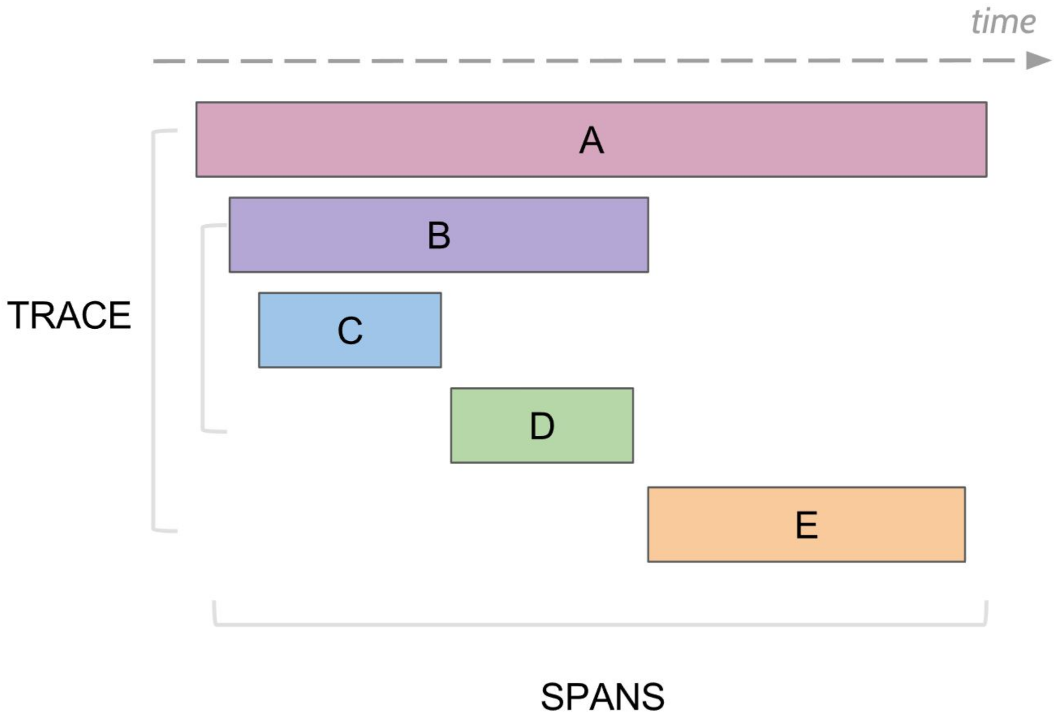
Distributed tracing



Logging between services

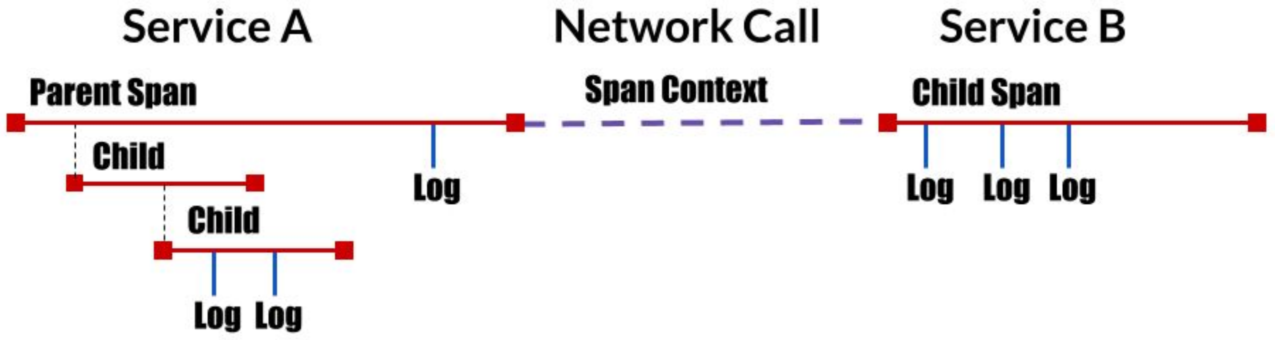
- Existing techniques for debugging and profiling begin to break down.
- No single service provides a complete picture.
- We need new tools to help us manage the real complexity of operating distributed systems

Opentracing terminology - Span/ Trace

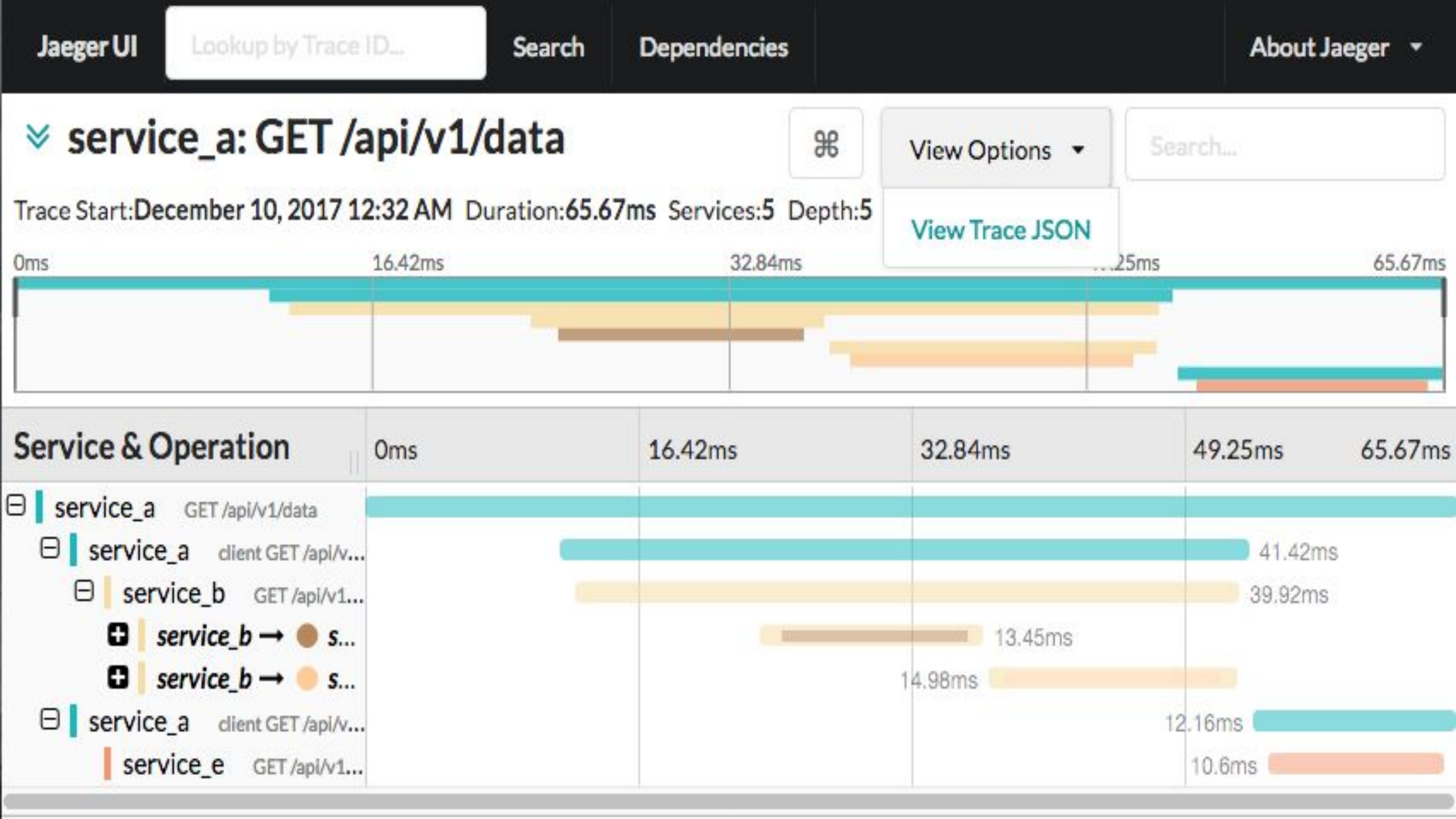


Logging between services

With distributed tracing, we can track requests as they pass through multiple services, emitting timing and other metadata throughout, and this information can then be reassembled to provide a complete picture of the application's behavior at runtime - buoyant



Mental model of distributed tracing - Opentracing



Jaeger UI | Lookup by Trace ID... | Search | Dependencies | About Jaeger

service_a: GET /api/v1/data

View Options

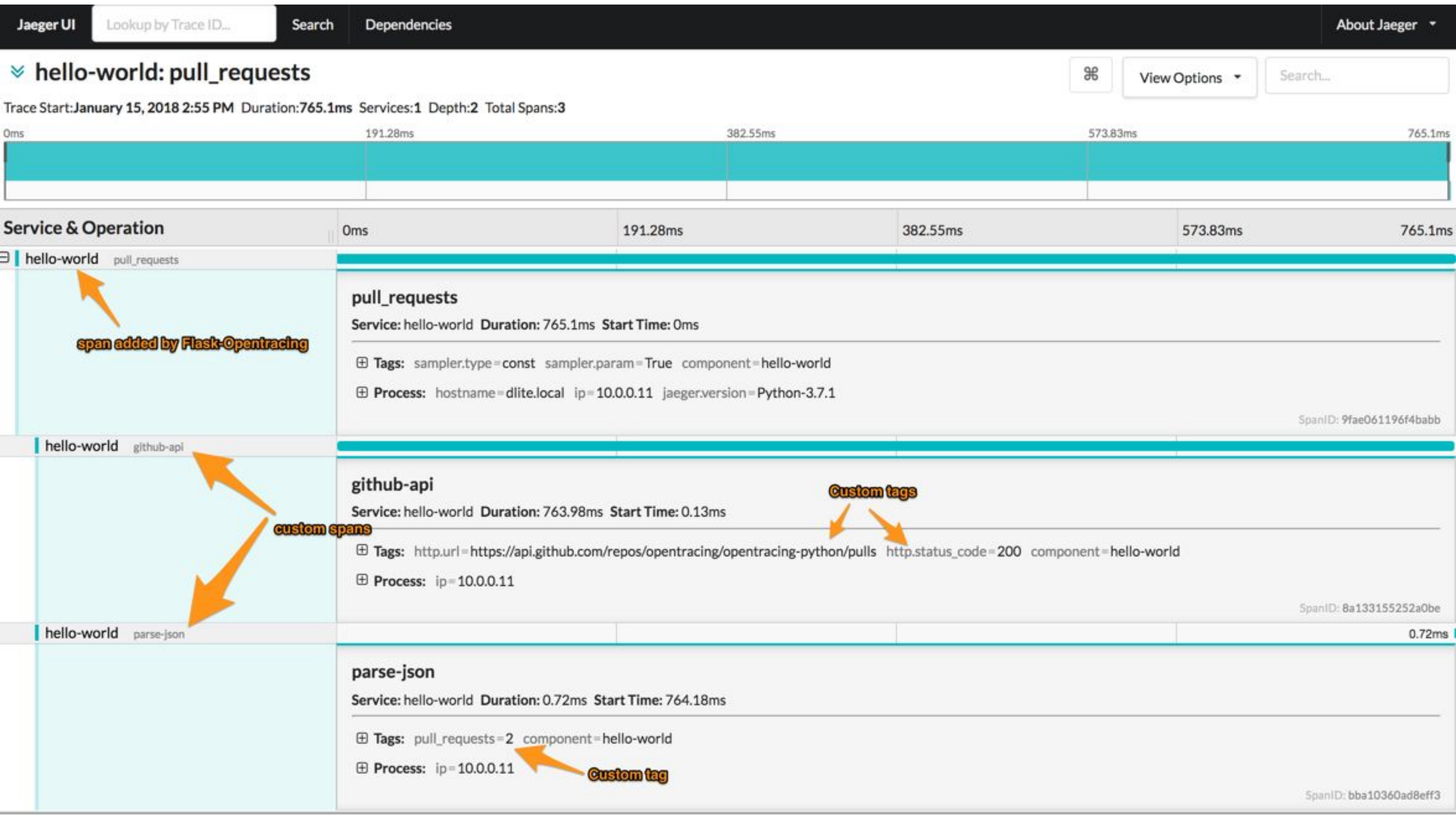
View Trace JSON

Search...

Trace Start: **December 10, 2017 12:32 AM** Duration: **65.67ms** Services: **5** Depth: **5**

Service & Operation | 0ms | 16.42ms | 32.84ms | 49.25ms | 65.67ms

```
⊟ service_a   GET /api/v1/data
  ⊟ service_a   client GET /api/v...        41.42ms
    ⊟ service_b   GET /api/v1...            39.92ms
      ⊞ service_b → s...                    13.45ms
      ⊞ service_b → s...                    14.98ms
  ⊟ service_a   client GET /api/v...        12.16ms
      service_e   GET /api/v1...            10.6ms
```



hello-world: pull_requests

Trace Start: **January 15, 2018 2:55 PM** Duration: **765.1ms** Services: **1** Depth: **2** Total Spans: **3**

hello-world pull_requests

span added by Flask-Opentracing

pull_requests

Service: hello-world **Duration:** 765.1ms **Start Time:** 0ms

⊞ **Tags:** sampler.type = const sampler.param = True component = hello-world

⊞ **Process:** hostname = dlite.local ip = 10.0.0.11 jaeger.version = Python-3.7.1

SpanID: 9fae061196f4babb

hello-world github-api

custom spans

github-api

Custom tags

Service: hello-world **Duration:** 763.98ms **Start Time:** 0.13ms

⊞ **Tags:** http.url = https://api.github.com/repos/opentracing/opentracing-python/pulls http.status_code = 200 component = hello-world

⊞ **Process:** ip = 10.0.0.11

SpanID: 8a133155252a0be

hello-world parse-json 0.72ms

parse-json

Service: hello-world **Duration:** 0.72ms **Start Time:** 764.18ms

⊞ **Tags:** pull_requests = 2 component = hello-world

⊞ **Process:** ip = 10.0.0.11

Custom tag

SpanID: bba10360ad8eff3

The future

The future of logging - on demand

Logging “right” in advance can be hard, new tools are trying to solve this issues:

1. Stackdriver (Mainly on Google services)
2. Trek10 (lambda+node only)
3. Rookout

The concept is neat, get the data YOU need, When you need it.

Questions?