Why you should be using *structured* logs

PyBay August 2019

Stefan Krawczyk

@stefkrawczyk
linkedin.com/in/skrawczyk

Try out Stitch Fix → goo.gl/Q3tCQ3

STITCH FIX

Outline:

What is Stitch Fix/Who am I?
What *logs* am I talking about?
What are *structured logs*?
Why are *structured logs* more fun?
Caveats with *structured logs*
How to implement a *structured logger*
Conclusion

Outline:

> **What is Stitch Fix/Who am I?**

What *logs* am I talking about?

What are *structured logs*?

Why are *structured logs* more fun?

Caveats with *structured logs*

How to implement a *structured logger*

Conclusion

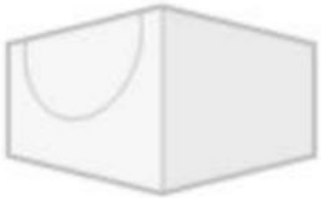
Personal Styling Service

Stitch Fix: Personal Styling Service

1

Create Your Style Profile

2



Get Five Hand-picked Items.

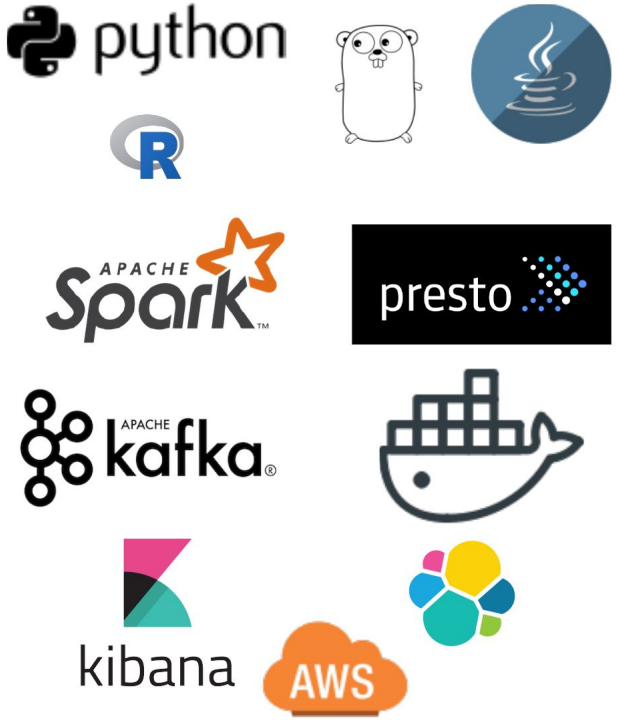
3

Keep What You Like.
Send Back the Rest.

Stitch Fix: Algorithms Org.

- Algorithms 110+ people
  - Data Scientists (~85%):
    - write their own ETL
    - manage & are on call for microservices
  - Data Platform:
    - avoids hand off
    - provides platforms, services, tools, libraries
    - steers best practices

- Over 100 microservices
  - Most owned by Data Scientists



Stitch Fix: Algorithms Org.

- Algorithms 110+ people
  - Data Scientists (~85%):
    - write their own ETL
    - manage & are on call for microservices
  - Data Platform: **< - - - - - - - - - - - - - - - - Manager of Algo. Dev. Platform Team.**
    - avoids hand off
    - provides platforms, services, tools, libraries
    - steers best practices

- Over 100 microservices
  - Most owned by Data Scientists

Outline:

What is Stitch Fix/Who am I?

> What *logs* am I talking about?

What are *structured logs*?

Why are *structured logs* more fun?

Caveats with *structured logs*

How to implement a *structured logger*

Conclusion

What *logs* am I talking about?

```
Aug 13 14:08:58 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.google.keystone.daemon[20144]): Endpoint has been activated through lega
launch(3) APIs. Please switch to XPC or bootstrap_check_in(): com.google.Keystone.Daemon.Administration
Aug 13 14:08:58 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.google.keystone.daemon[20144]): Endpoint has been activated through lega
launch(3) APIs. Please switch to XPC or bootstrap_check_in(): com.google.Keystone.Daemon.UpdateEngine
Aug 13 14:11:30 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.jamfsoftware.task.Every 15 Minutes[20014]): Service exited with abnormal
code: 1
Aug 13 14:19:47 ip-10-8-0-122 syslogd[64]: ASL Sender Statistics
Aug 13 14:20:45 ip-10-8-0-122 AOUDownloadCount[20978]: ERROR|AOUDownloadCount.m|376L|Error:AOUDownloadCount::sendDownloadCountInfo:g
DownloadCountInfo failed.
Aug 13 14:29:26 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.jamfsoftware.task.Every 15 Minutes[21148]): Service exited with abnormal
code: 1
Aug 13 14:30:04 ip-10-8-0-122 syslogd[64]: ASL Sender Statistics
Aug 13 14:30:58 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.apple.quicklook[21738]): Endpoint has been activated through legacy
launch(3) APIs. Please switch to XPC or bootstrap_check_in(): com.apple.quicklook
Aug 13 14:35:23 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.apple.imfoundation.IMRemoteURLConnectionAgent): Unknown key for integer:
_DirtyJetsamMemoryLimit
Aug 13 14:42:27 ip-10-8-0-122 syslogd[64]: ASL Sender Statistics
Aug 13 14:46:31 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.jamfsoftware.task.Every 15 Minutes[24285]): Service exited with abnormal
code: 1
Aug 13 14:47:20 ip-10-8-0-122 com.apple.xpc.launchd[1] (com.apple.xpc.launchd.domain.user.502): Service
"com.apple.xpc.launchd.unmanaged.loginwindow.123" tried to register for endpoint "com.apple.tsm.uiserver" already registered by owne
com.apple.SystemUIServer.agent
```

What *logs* am I talking about?

- Service (Application) logs
  - e.g. created via:

```
import logging
logger = logging.getLogger(__name__)
...
logger.info('Computing %s for client_id: %s, shipment_id: %s', item_id, client_id, shipment_id)
```

- Service Level Agreement: Best effort

- Apps are written by many different developers.
  - logs are pushed to a central store.

Example use cases for logs

Debugging/Monitoring

- want to find all logs relating to a particular thing, e.g. client_id
- want a dashboard to automatically capture new app logs/values
- are paged, and have to figure out what's going on
- want to find all stack traces

Analysis/Reporting

- extract custom metrics
- trend analysis/reporting over time

Examples on how you likely consume logs

- Key word search
  - `grep “client”`
- Regular Expressions
  - `egrep “client[_id: ]+(\d+)”`
- Parsers/analyzers + query interface.
  - e.g. splunk, elasticsearch, etc.

Question:

Who here likes *using* logs?

Outline:

What is Stitch Fix/Who am I?

What *logs* am I talking about?

> What are *structured logs*?

Why are *structured logs* more fun?

Caveats with *structured logs*

How to implement a *structured logger*

Conclusion

What are *structured logs*?

Idea - take:

```
2019-07-10 18:46:34,501 INFO logging_talk_code __main__ 18 Computing 10001 for client_id 1 shipment_id
2
```

Turn it into a machine friendlier format:

```
{
    "message": "Computing 10001 for client_id 1 shipment_id 2",
    "payload": {
      "client_id": 1,
      "shipment_id": 2,
      "item_id": 10001
    },
    "metadata": {
      "code": {
        "file_url": "/a/url/code.py",
        "line_number": 186,
        "file_name": "code.py",
```

What are *structured logs*?

Developers already implicitly provide structure:

```
logger.info('Computing %s for client_id: %s, shipment_id: %s', item_id, client_id, shipment_id)
```

or

```
logger.info(f'Computing {item_id} for client_id: {client_id}, shipment_id: {shipment_id}')
```

or

```
logger.info('Computing %(item_id)s for client_id: %(client_id)s shipment_id: %(shipment_id)s',
            {'client_id': client_id, 'item_id': item_id, 'shipment_id': shipment_id})
```

What are *structured logs*?

Connecting developer intent:

```
logger.info('Computing %s for client_id: %s, shipment_id: %s', item_id, client_id, shipment_id)
```

and propagating it to the output better:

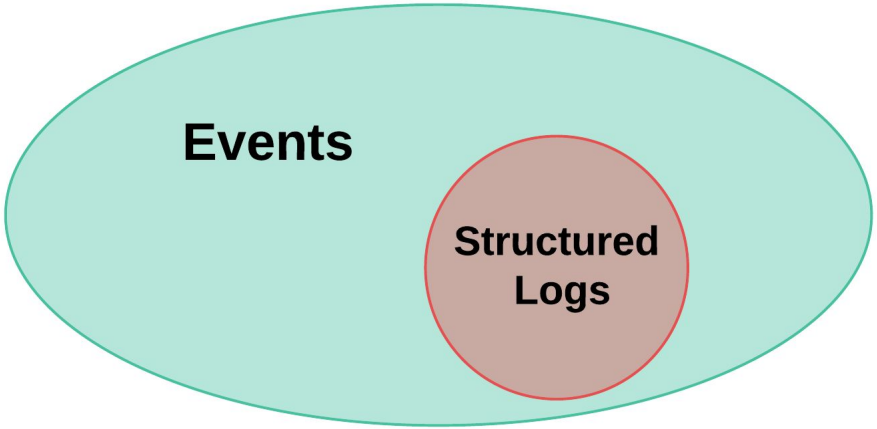
Intent is lost:

```
Computing 10001 for client_id 1 shipment_id
```

Intent is retained:

```
{
    "message": "Computing 10001 for client_id 1
shipment_id 2",
    "payload": {
      "client_id": 1,
      "shipment_id": 2,
      "item_id": 10001
    },
    "metadata": {
       "code": {
        "file_url": "/a/url/path/code.py",
        "line_number": 186,
```

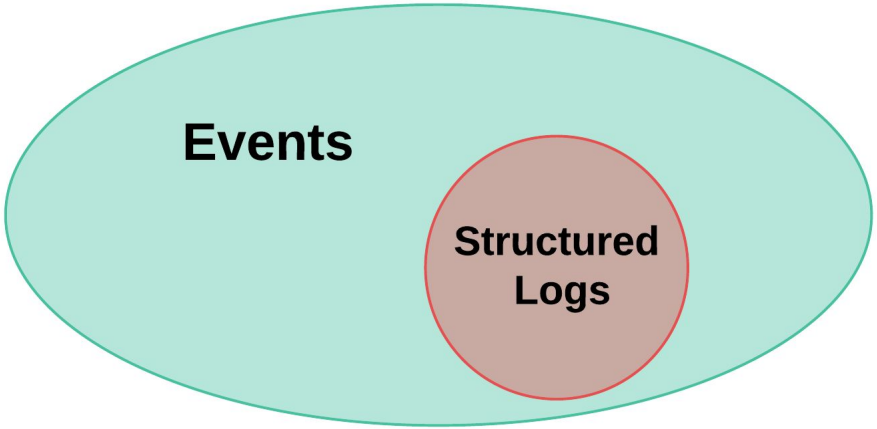
What are *structured logs*?



Possible output formats:

- Thrift
- Protobuf
- Avro
- JSON
- CSV
- your own structure

What are *structured logs*?



Possible output formats:

- Thrift
- Protobuf
- Avro
- JSON
- CSV
- your own structure

You should consider them part of your event ecosystem.

What are *structured logs* @ Stitch Fix?

Key part:

- developer intent passed through as dictionary:

```
logger.info('Some message optional %(interpolation)s', log_payload)
```

Format:

- We use JSON.

Why JSON?

- Lots of tools read/write JSON.
- It's easy to implement and maintain.

What are *structured logs* @ Stitch Fix?

In addition to developer data, our JSON Structure includes:

- file name
- function name
- line number
- module
- process id
- metadata about where the code is being run
- stack traces
- contextual information, e.g. request_id

What are *structured logs* @ Stitch Fix?

In addition to developer data, our JSON Structure includes:

- file name
- function name
- line number
- module
- process id
- metadata about where the code is being run
- stack traces
- contextual information, e.g. request_id

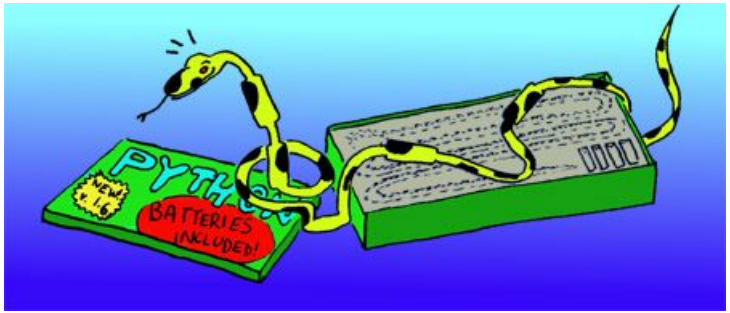
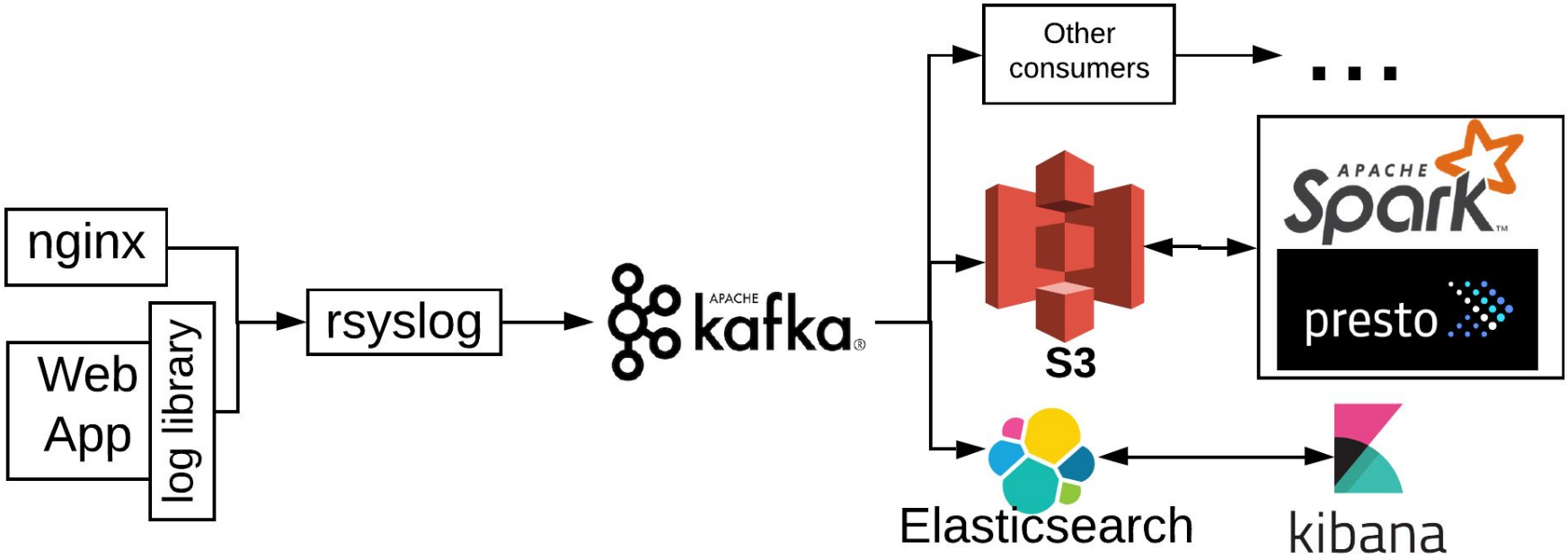
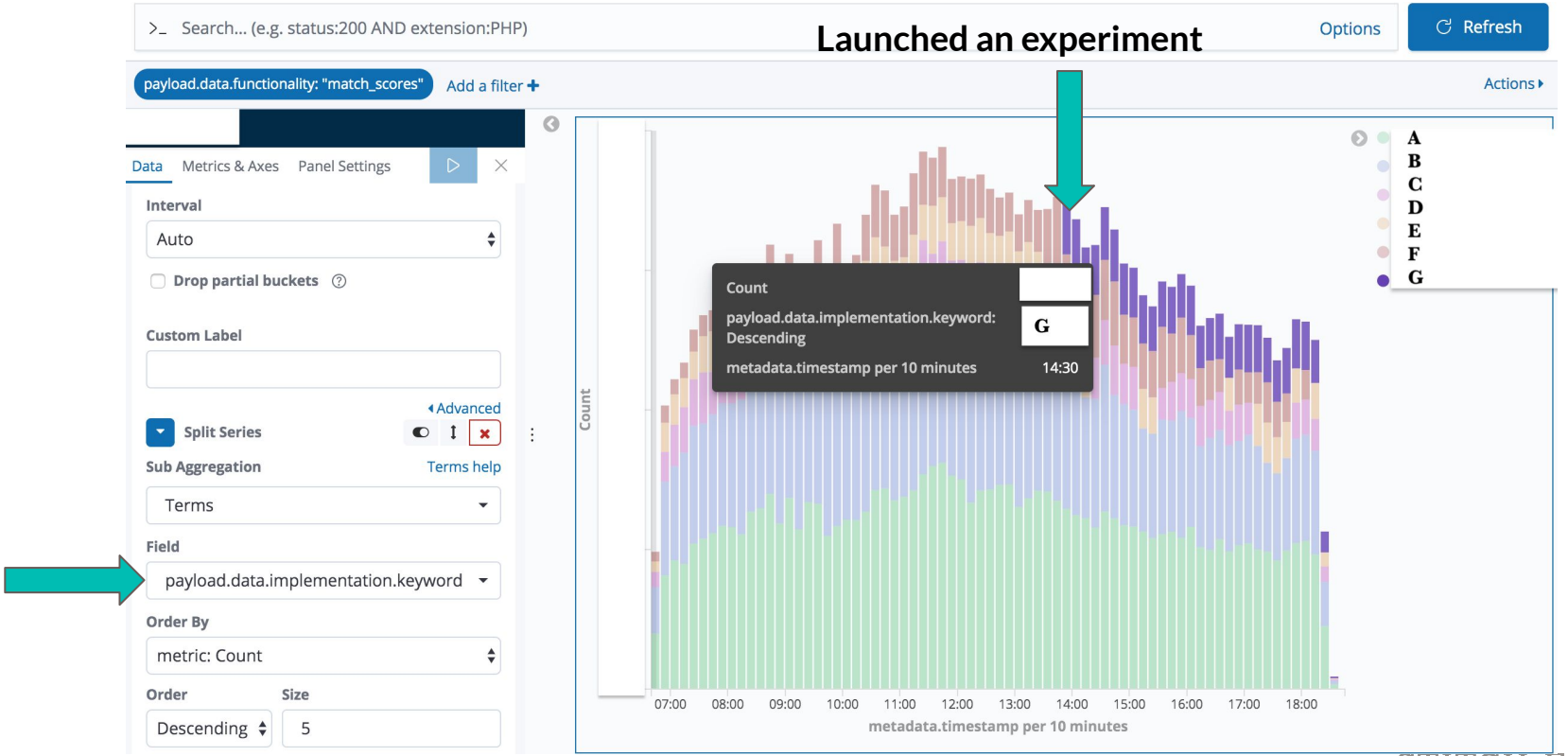


image:https://commons.wikimedia.org/wiki/File:Python_batteries_included.jpg

What are *structured logs* @ Stitch Fix?



Structured logs @ Stitch Fix: Example Usage



Outline:
What is Stitch Fix/Who am I?
What *logs* am I talking about?
What are *structured logs*?
> Why are *structured logs* more fun?
Caveats with *structured logs*
How to implement a *structured logger*
Conclusion

Why are *structured logs* more fun?

Why?

- TL;DR: easier to consume.

What's easier to use/maintain?

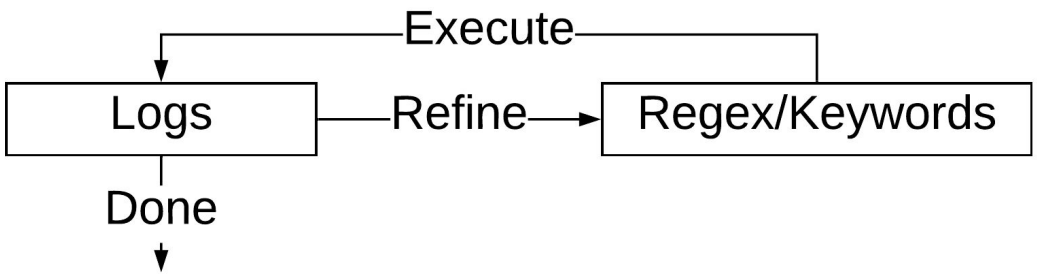
- key word search, regex & custom parsers
- or machine friendly field lookups


```
{
    "message": "Computing 10001 for client_id 1
shipment_id 2",
    "payload": {
      "client_id": 1,
      "shipment_id": 2,
      "item_id": 10001
     },
     "metadata": {
       "code": {
        "file_url": "/a/url/code.py",
        "line_number": 186,
        "file_name": "code.py",
        "module": "code",
        "function_name": "up_next"
      },
      "instance": {
        "ami_id": "XXXX",
        "id": "XXXX",
        "type": "c4.2xlarge",
        "tags": {
          "aws:autoscaling:groupName":
```

Why are *structured logs* more fun?

If it's faster to use, it's more fun to use:

Regular Logs:

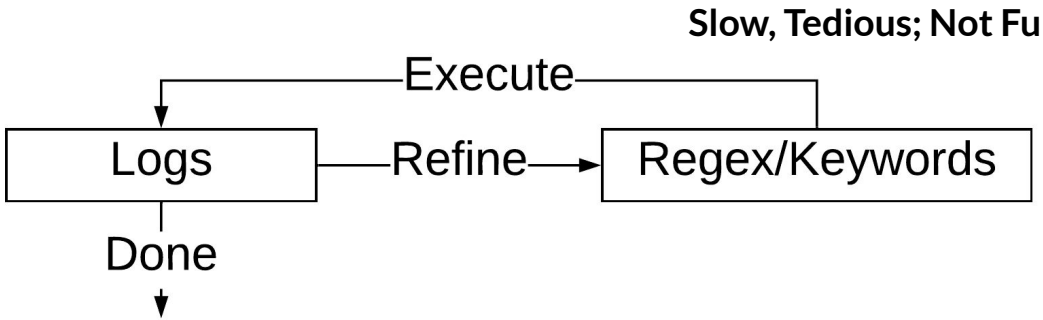


Structured Logs:

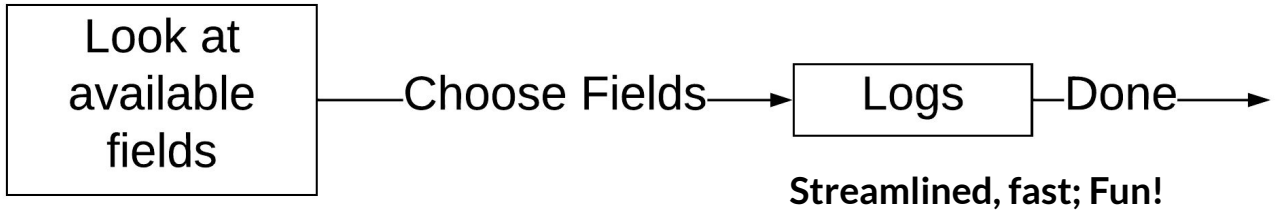
Why are *structured logs* more fun?

If it's faster to use, it's more fun to use:

Regular Logs:



Structured Logs:



Why are *structured logs* more fun?

Conjecture:

- Happy developers are more productive developers!

“I love logging now. It’s so useful”

- A Stitch Fix Colleague

Why are *structured logs* more fun?

It builds a leverageable ecosystem:

- **Developer knowledge transfer**
  - **much easier to pick up, use, and evangelize.**

Leverage: Developer Knowledge Transfer

- **one *incentivized* way to write a logline!**
- **only *one* format that logs come out in!**

e.g.

```
logger.info('some text %(var1)s, %(var2)s', {'var1': value1, 'var2': value2})
```

will be:

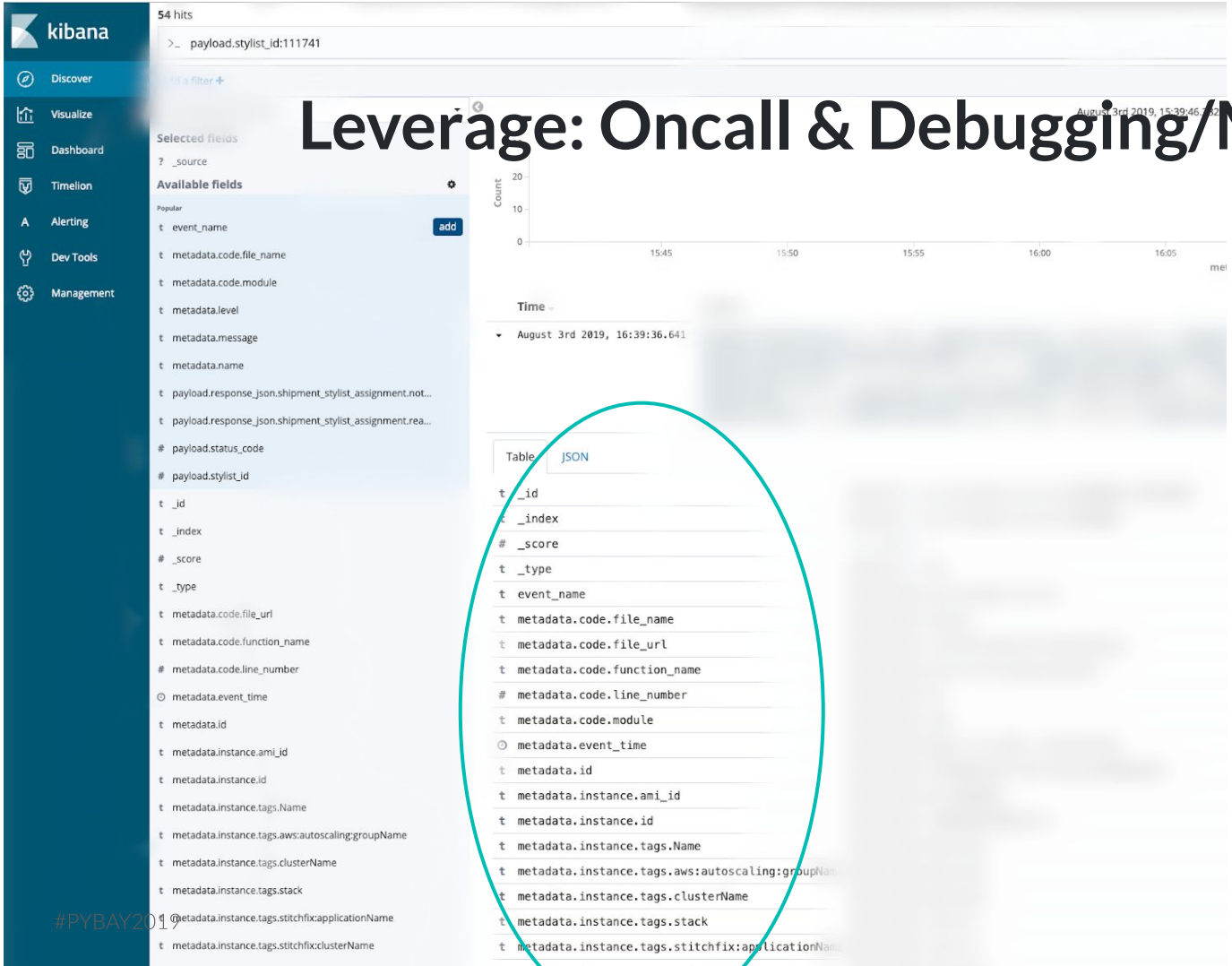
- a single payload we'll know the structure of, not some arbitrary structure. Once learned, it's reusable.

Why are *structured logs* more fun?

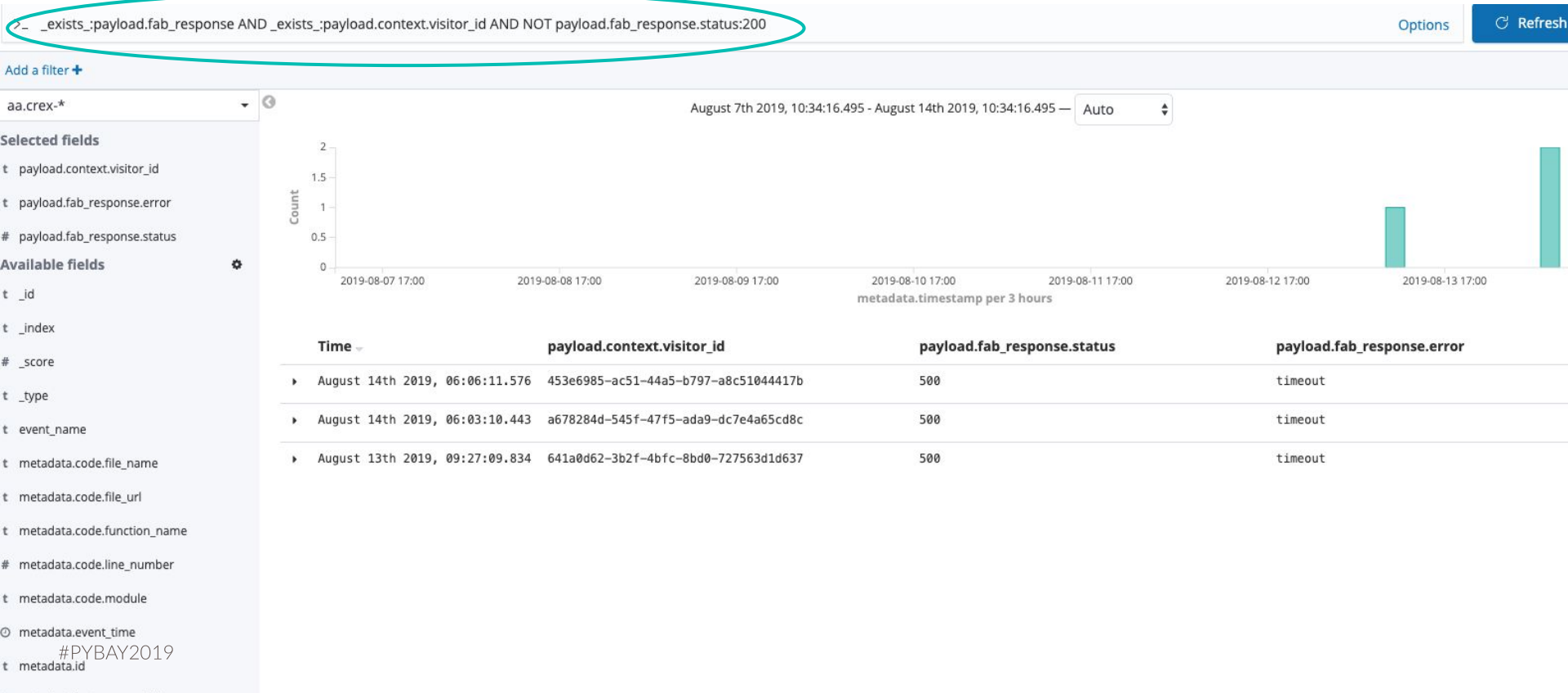
It builds a leverageable ecosystem:

- Developer knowledge transfer
  - much easier to pick up, use, and evangelize.
- **Oncall debugging/monitoring:**
  - **inspecting & using logs is easier since it's in a structured format (e.g. stacktraces)**

Leverage: Oncall & Debugging/Monitoring



Leverage: Oncall & Debugging/Monitoring



Leverage: Oncall & Debugging/Monitoring

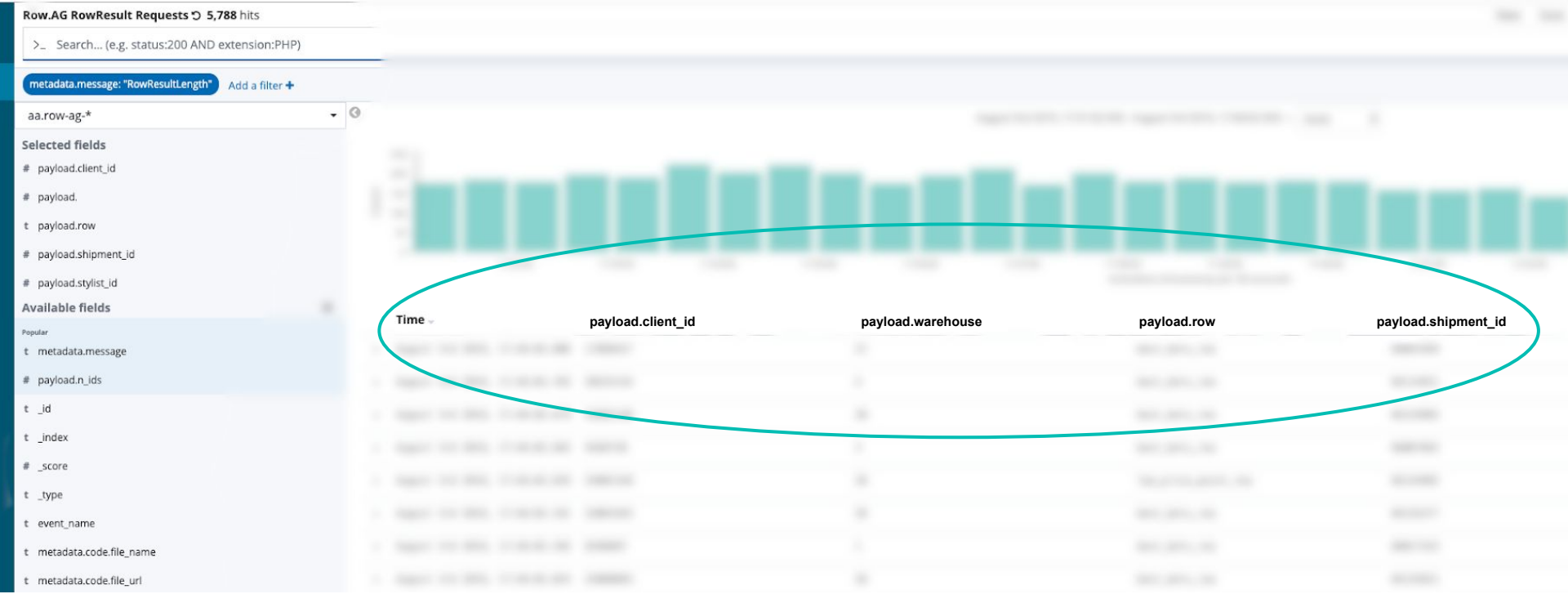
E.g.

- Multi-line stack traces are their own field!

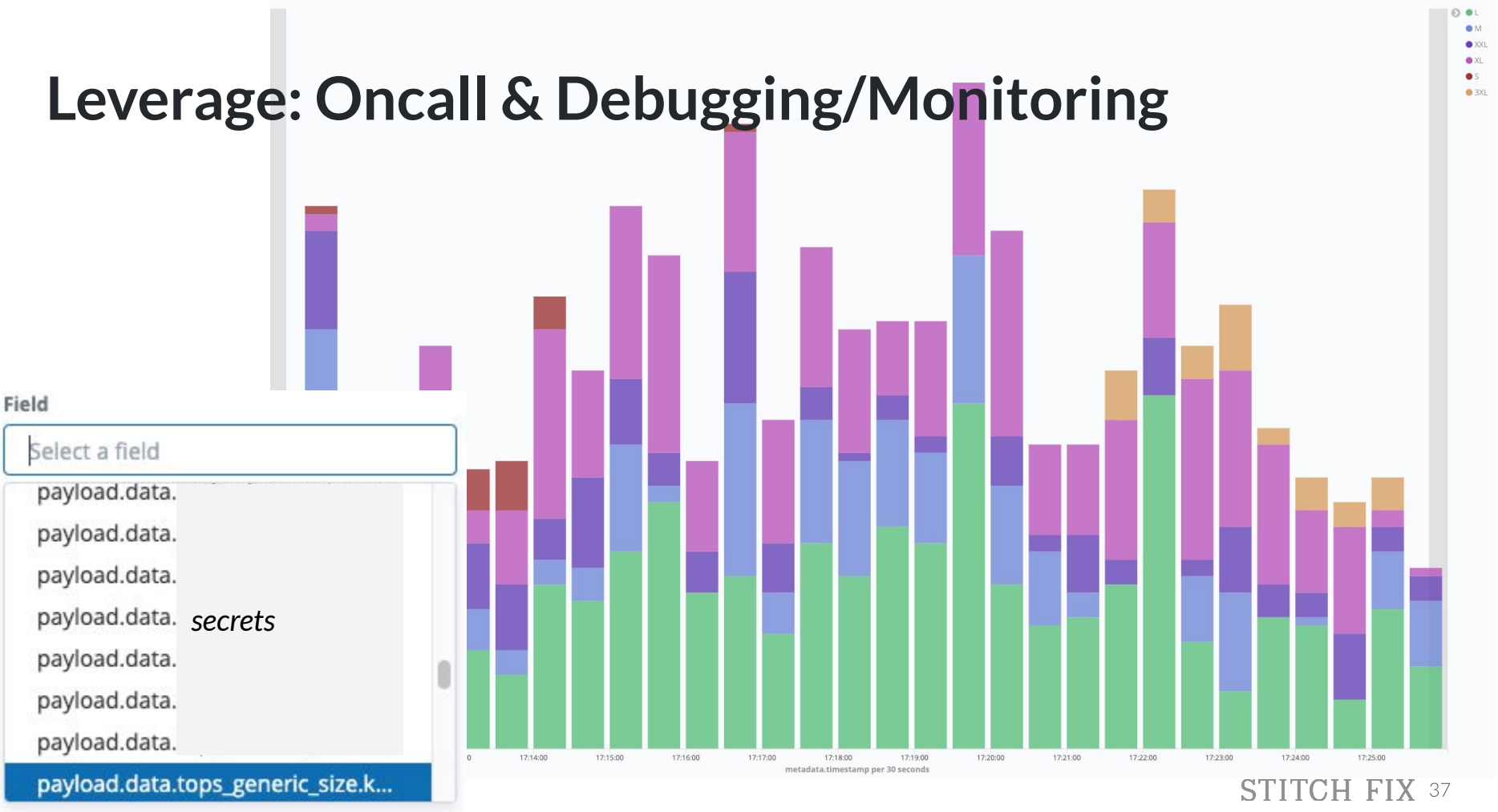
```
t  metadata.stack_trace

Exception while handling request
Traceback (most recent call last):
  File "/stitchfix/z-pyenv/pyenv/versions/3.6.7/lib/python3.6/site-packages/sanic/app.py", line 503, in handle_requ
est
    response = await response
  File "/stitchfix/z-pyenv/pyenv/versions/3.6.7/lib/python3.6/site-packages/khan/server.py", line 231, in handler
    output = await output
  File "
    debug
  File "
    client
  File "
    value
  File "
    respon
  File "
    retu
  File
    return await self._request('post', url, *args, **kwargs)
  File "/stitchfix/z-pyenv/pyenv/versions/3.6.7/lib/python3.6/site-packages/khan/clients/http.py", line 134, in _re
quest
    response=wrapped_resp, request=request_info, status=resp.status
khan.clients.http.HTTPClientError: Error 500: {"error":"MissingClientFeature","stacktrace":"Traceback (most recent
 call last):\n  File \"\/stitchfix\/z-pyenv\/pyenv\/versions\/3.6.7\/lib\/python3.6\/site-packages\/sanic\/app.py
\", line 503, in handle_request\n    response = await response\n  File \"\/stitchfix\/z-pyenv\/pyenv\/versions\/3.
6.7\/lib\/python3.6\/site-packages\/khan\/server.py\", line 231, in handler\n    output = await output\n  File \"\/
stitchfix\/portal\/
```

Leverage: Oncall & Debugging/Monitoring



Leverage: Oncall & Debugging/Monitoring



Why are *structured logs* more fun?

It builds a leverageable ecosystem:

- Developer knowledge transfer
  - much easier to pick up, use, and evangelize.
- Oncall debugging/monitoring:
  - inspecting & using logs is easier since it's in a structured format (e.g. stacktraces)
- **Easy to ingest/use in multiple places/pieces of infrastructure**
  - **e.g. elasticsearch, presto, ETL, custom reports**

Leverage: Easy to reuse in multiple places

- For ingestion into infrastructure, setup is simpler.
- Easy to create tables in the data warehouse from same logs that go to elasticsearch:

```
with m_app_traces AS (
  SELECT
     json_extract_scalar(m.payload, '$.metadata.http_headers["stitchfix-request-id"]') as request_id,
     json_extract_scalar(m.payload, '$.payload.data.functionality') as m_functionality,
     json_extract_scalar(m.payload, '$.payload.data.context.client_id') as client_id,
     json_extract_scalar(m.payload, '$.payload.data.context.shipment_id') as shipment_id,
     json_extract_scalar(m.payload, '$.payload.data.context.stylist_id') as stylist_id
  FROM
     kafkatopic.secret_app_logs m
  WHERE
     m.date = '20190817'
),
...
```

Leverage: Easy to reuse in multiple places

Also easy to pull them into python and do analysis with pandas...

```
def create_df_for_analysis(lines: List[str]) -> int:
    """Counts number of errors in log lines"""
    df_base = {'level': [], 'message': [], 'client_id': [], 'timestamp': []}
    for line in lines:
        json_dict = json.loads(line)  # We know the structure so easy to analyze
        df_base['timestamp'] = json_dict['timestamp']
        df_base['level'] = json_dict['level']
        df_base['message'] = json_dict['message']
        df_base['client_id'] = json_dict['payload'].get('client_id')
    df = pd.DataFrame(df_base)
    df.index = df['timestamp']
    return df
```

Why are *structured logs* more fun?

It builds a leverageable ecosystem:

- Developer knowledge transfer
  - much easier to pick up, use, and evangelize.
- Oncall debugging/monitoring:
  - inspecting & using logs is easier since it's in a structured format (e.g. stacktraces)
- Easy to ingest/use in multiple places/pieces of infrastructure
  - e.g. elasticsearch, presto, ETL, custom reports
- **Can tie other systems together more easily:**
  - **e.g. nginx, web apps, ETLs, etc.; *can use/join on the same "key" names.***

Leverage: tie systems together

If all logs are structured:

- easier to tie systems together.

E.g. if we want to tie nginx logs with app logs

```
with app1 AS (
  SELECT
     m.event_timestamp,
     json_extract_scalar(m.payload, '$.query_params.client_id') as client_id,
     json_extract_scalar(m.payload, '$.response_time') as response_time
  FROM
     kafkatopic.nginx_logs m
  WHERE
     m.date = '20190817'
),
app2 AS (
  SELECT
     m.event_timestamp,
     json_extract_scalar(m.payload, '$.payload.client_id') as client_id,
     json_extract_scalar(m.payload, '$.payload.score') as score
  FROM
     kafkatopic.app1_logs m
  WHERE
     m.date = '20190817'
)
SELECT
     a1.client_id,   a1.event_timestamp,
     a1.response_time,
     a2.score
  FROM
     app1 a1
 JOIN
     app2 a2
  ON
     a1.client_id = a2.client_id
```

Outline:
What is Stitch Fix/Who am I?
What *logs* am I talking about?
What are *structured logs*?
Why are *structured logs* more fun?
> Caveats with *structured logs*
How to implement a *structured logger*
Conclusion

Untyped -> Typed

```
logger.info('Computing scores for client_id: %(client_id)s shipment_id: %(shipment_id)s',
            {'client_id': client_id, 'shipment_id': shipment_id})
```

Some downstream consumers require types to be declared.

- e.g. elasticsearch, ETLs

Untyped -> Typed

```
logger.info('Computing scores for client_id: %(client_id)s shipment_id: %(shipment_id)s',
            {'client_id': client_id, 'shipment_id': shipment_id})
```

Some downstream consumers require types to be declared.

- e.g. elasticsearch, ETLs

Problems:

- Application developer changes type of a value.
  - e.g. from int to string , e.g. 123 to “abc”.
- What is empty?
  - None, “”, 0, don't pass it?

Naming

```
logger.info('Computing scores for client_id: %(client_id)s shipment_id: %(shipment_id)s',
            {'client_id': client_id, 'shipment_id': shipment_id})
```

- Keys/structure needs to be consistent

Naming

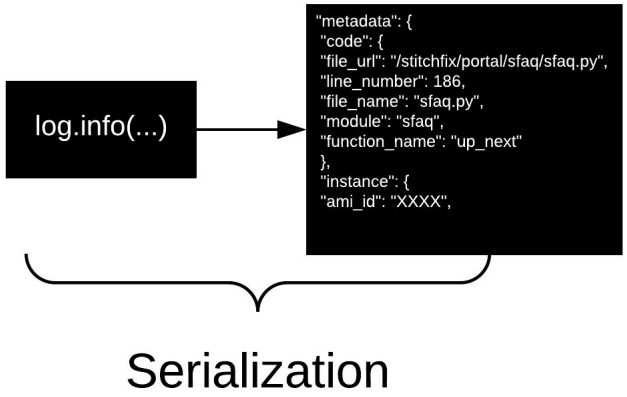
```
logger.info('Computing scores for client_id: %(client_id)s shipment_id: %(shipment_id)s',
            {'client_id': client_id, 'shipment_id': shipment_id})
```

- Keys/structure needs to be consistent
  - either depend on developers
  - create a constants module
    - In a monolith this is important.

Serialization

Problem:

- Object serializability
  - e.g. datetime.
- Speed
  - need to consider your context.

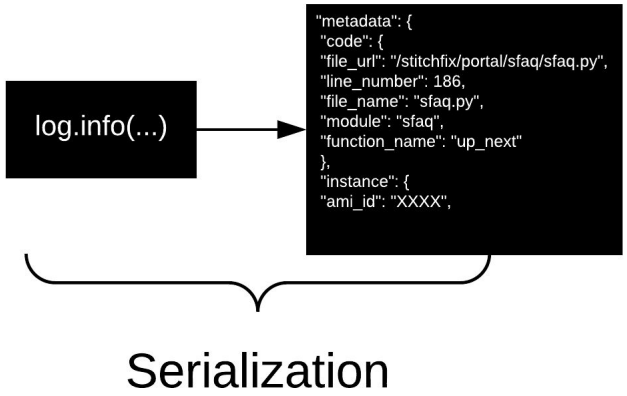


[*] https://docs.python.org/3/howto/logging-cookbook.html#dealing-with-handlers-that-block

Serialization

Problem:

- Object serializability
  - e.g. datetime.
- Speed
  - need to consider your context.



Options for better speed:

- ujson is pretty fast.
- QueueHandler + Listener* could be an option.

[*] https://docs.python.org/3/howto/logging-cookbook.html#dealing-with-handlers-that-block

Human Readable Logs

Humans like reading:

```
Computing scores for client_id 1 shipment_id 2
```

Over:


```
"metadata": {
      "code": {
        "file_url": "/a/url/path/code.py",
        "line_number": 186,
        "file_name": "code.py",
        "module": "code",
        "function_name": "up_next"
      },
      "instance": {
        "ami_id": "XXXX",
        "id": "XXXX",
        "type": "c4.2xlarge",
        "tags": {
          "aws:autoscaling:groupName": "k-prod",
          "stack": "flotilla",
          "stitchfix:clusterName": "k-prod",
          "clusterName": "k-prod",
          "stitchfix:applicationName": "our_app",
          "name": "k-prod"
        }
      },
```


[*] https://docs.python.org/3/howto/logging-cookbook.html#logging-to-multiple-destinations

Human Readable Logs

Humans like reading:

```
Computing scores for client_id 1 shipment_id 2
```

Over:


```
"metadata": {
      "code": {
        "file_url": "/a/url/path/code.py",
        "line_number": 186,
        "file_name": "code.py",
        "module": "code",
        "function_name": "up_next"
      },
      "instance": {
        "ami_id": "XXXX",
        "id": "XXXX",
        "type": "c4.2xlarge",
        "tags": {
          "aws:autoscaling:groupName": "k-prod",
          "stack": "flotilla",
          "stitchfix:clusterName": "k-prod",
          "clusterName": "k-prod",
          "stitchfix:applicationName": "our_app",
          "name": "k-prod"
        }
      },
```


Options:

- Multiple handlers*
- Write tooling
  - “json_cat”
  - tail -f my.log | jq ...

[*] https://docs.python.org/3/howto/logging-cookbook.html#logging-to-multiple-destinations

Outline:
What is Stitch Fix/Who am I?
What *logs* am I talking about?
What are *structured logs*?
Why are *structured logs* more fun?
Caveats with *structured logs*
> How to implement a *structured logger*
Conclusion

First: Python Logging Module

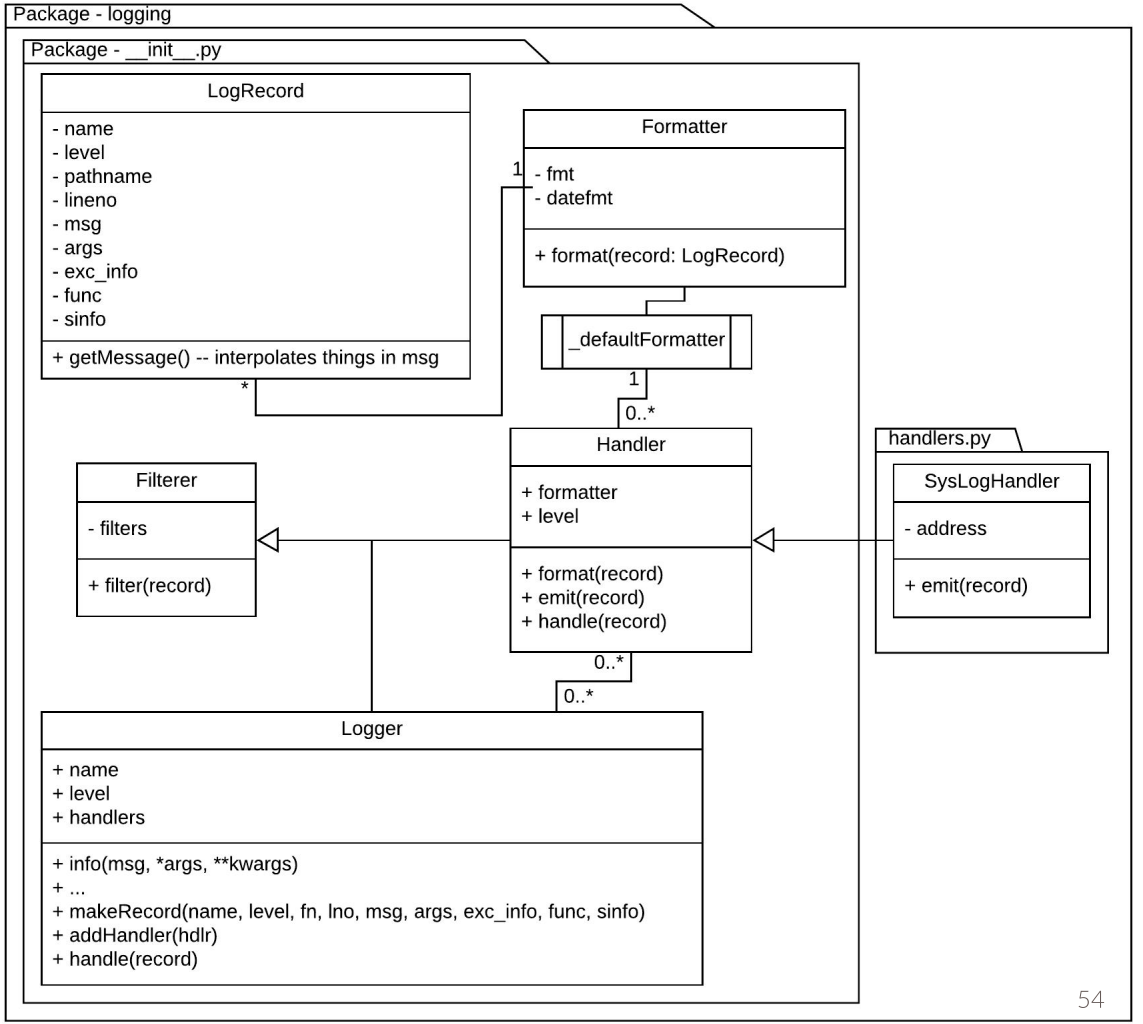
Python Logging Module 3.6+

Most people:

- use base Logger class
- Pass in a format
- Choose log handler of choice

For more details:

https://docs.python.org/3/howto/logging.html
https://docs.python.org/3/howto/logging-cookbook.html



Python Logging Module 3.6+

Logger Class:

- interface to developers
- makes LogRecord
- delegates to handlers

```
class Logger(Filterer):
    """...
    def __init__(self, name, level=NOTSET):
        """
        Initialize the logger with a name and an optional level.
        """
        Filterer.__init__(self)
        self.name = name
        self.level = _checkLevel(level)
        self.parent = None
        self.propagate = True
        self.handlers = []
        self.disabled = False
        self._cache = {}
    def info(self, msg, *args, **kwargs):...

    def warning(self, msg, *args, **kwargs):...

    def warn(self, msg, *args, **kwargs):...

    def error(self, msg, *args, **kwargs):...

    def exception(self, msg, *args, exc_info=True, **kwargs):...
```

```
    def makeRecord(self, name, level, fn, lno, msg, args, exc_info,
                   func=None, extra=None, sinfo=None):
        """
        A factory method which can be overridden in subclasses to create
        specialized LogRecords.
        """
        rv = _logRecordFactory(name, level, fn, lno, msg, args, exc_info, func,
                               sinfo)
        if extra is not None:
            for key in extra:
                if (key in ["message", "asctime"]) or (key in rv.__dict__):
                    raise KeyError("Attempt to overwrite %r in LogRecord" % key)
                rv.__dict__[key] = extra[key]
        return rv
```

Python Logging Module 3.6+

LogRecord:

- encapsulates a log

```
class LogRecord(object):
    """A LogRecord instance represents an event being logged...."""
    def __init__(self, name, level, pathname, lineno,
                 msg, args, exc_info, func=None, sinfo=None, **kwargs):...

    def __str__(self):
        return '<LogRecord: %s, %s, %s, %s, "%s">'%(self.name, self.levelno,
            self.pathname, self.lineno, self.msg)

    __repr__ = __str__

    def getMessage(self):
        """
        Return the message for this LogRecord.

        Return the message for this LogRecord after merging any user-supplied
        arguments with the message.
        """
        msg = str(self.msg)
        if self.args:
            msg = msg % self.args
        return msg
```

Python Logging Module 3.6+

Handler:

- base class
- format()
- emit() -> destination

```
class Handler(Filterer):
    """
    Handler instances dispatch logging events to specific destinations.

    The base handler class. Acts as a placeholder which defines the Handler
    interface. Handlers can optionally use Formatter instances to format
    records as desired. By default, no formatter is specified; in this case,
    the 'raw' message as determined by record.message is logged.
    """
    def __init__(self, level=NOTSET):...

    def get_name(self):...

    def set_name(self, name):...

    name = property(get_name, set_name)

    def createLock(self):...

    def acquire(self):...

    def release(self):...

    def setLevel(self, level):...

    def format(self, record):
        """
        Format the specified record.

        If a formatter is set, use it. Otherwise, use the default formatter
        for the module.
        """
        if self.formatter:
            fmt = self.formatter
        else:
            fmt = _defaultFormatter
        return fmt.format(record)

    def emit(self, record):
        """
        Do whatever it takes to actually log the specified logging record.

        This version is intended to be implemented by subclasses and so
        raises a NotImplementedError.
        """
        raise NotImplementedError('emit must be implemented '
                                  'by Handler subclasses')
```

Python Logging Module 3.6+

Formatter:

- format(record) -> str

```
class Formatter(object):
    """Formatter instances are used to convert a LogRecord to text...."""

    converter = time.localtime

    def __init__(self, fmt=None, datefmt=None, style='%'):...

    default_time_format = '%Y-%m-%d %H:%M:%S'
    default_msec_format = '%s,%03d'

    def format(self, record):
        """
        Format the specified record as text.

        The record's attribute dictionary is used as the operand to a
        string formatting operation which yields the returned string.
        Before formatting the dictionary, a couple of preparatory steps
        are carried out. The message attribute of the record is computed
        using LogRecord.getMessage(). If the formatting string uses the
        time (as determined by a call to usesTime(), formatTime() is
        called to format the event time. If there is exception information,
        it is formatted using formatException() and appended to the message.
        """
        record.message = record.getMessage()
        if self.usesTime():
            record.asctime = self.formatTime(record, self.datefmt)
        s = self.formatMessage(record)
        if record.exc_info:
            # Cache the traceback text to avoid converting it multiple times
            # (it's constant anyway)
            if not record.exc_text:
                record.exc_text = self.formatException(record.exc_info)
        if record.exc_text:
            if s[-1:] != "\n":
                s = s + "\n"
            s = s + record.exc_text
        if record.stack_info:
            if s[-1:] != "\n":
                s = s + "\n"
            s = s + self.formatStack(record.stack_info)
        return s
```

TL;DR: Logging module (3.6+)

- **LogRecord** encapsulates what you want to log.
- **Formatter**:
  - formats the LogRecord object.
- **Handler**:
  - Uses the formatter.
  - Dumps the output to a destination.
- **Logger**:
  - Interface that developers use in their code to log events.
  - Makes the LogRecord object.

https://docs.python.org/3/howto/logging.html
https://docs.python.org/3/howto/logging-cookbook.html

Second: Options for implementation

Very happy we're in Python

- **args* & ***kwargs* are really magical, compared to other languages

```
def info(self, msg, *args, **kwargs):
```

- Allows us to reuse underlying python logging library!
- Why? Two pathways:
  - args[0] can be a dict. (see LogRecord constructor)
  - *extra=* keyword available for *extra* things. (see Logger.makeRecord())

Options

Lots of ways to implement:

- Very dependent on output format

Options

Lots of ways to implement:

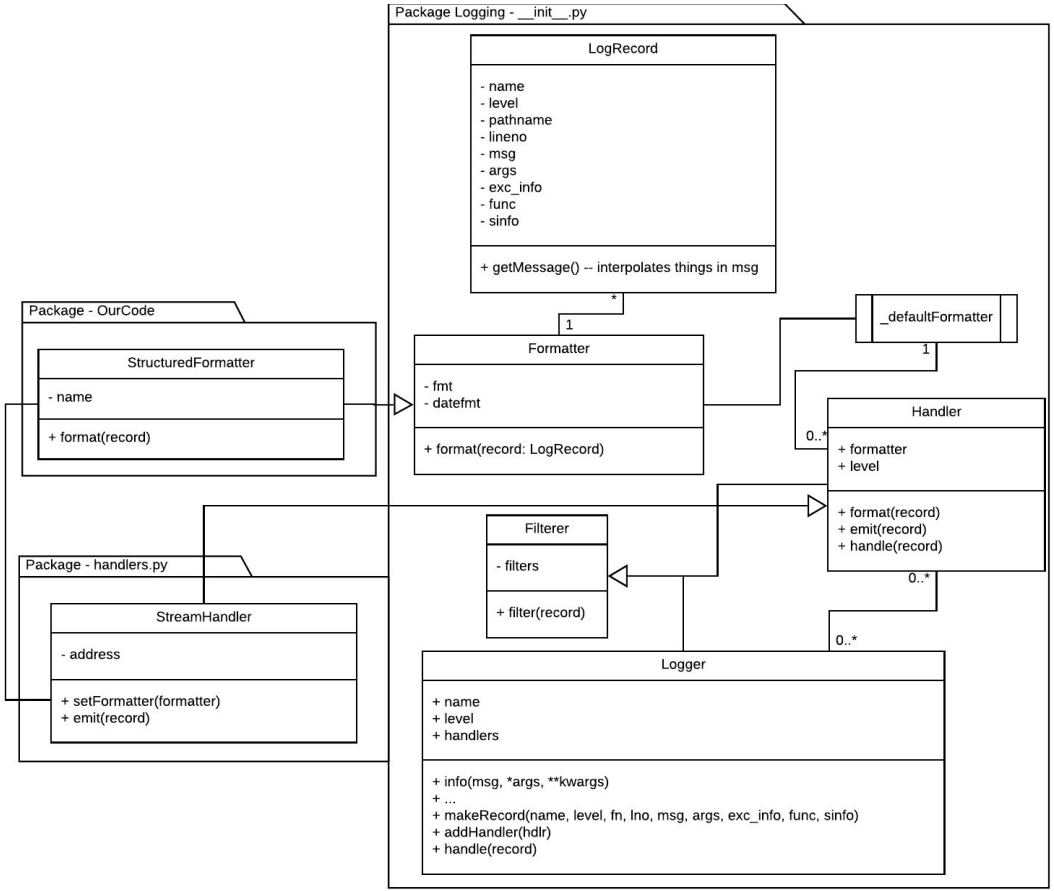
- Very dependent on output format

E.g.

- Implement own Logger, Handler, Formatter
  - Override specific methods
    - e.g. format(), makeRecord(), emit(), etc.

JSON *structured logger*

- JSON is a string in the end.
  - Simplest is to create own formatter.
    - override **format()**
- For more esoteric needs:
  - Custom Handler
    - override **format()/emit()**
  - Custom Logger & Handler
    - override **makeRecord()**
    - override **format()**



Example code

Creating a custom formatter and hooking it up to the console.

Code

```
class PyBayFormatter(logging.Formatter):
    """Implementation of JSON structured logging that works for most handlers."""

    def format(self, record: logging.LogRecord) -> str:
        """Overrides parent format function.

        :param record: logging.LogRecord object
        :return: JSON string
        """
        payload = self.make_structured_dict(record)
        if hasattr(record, 'exc_info') and record.exc_info:  # adds stack traces!
                payload['stack_trace'] = self.formatException(record.exc_info)
        return json.dumps(payload)

    @staticmethod
    def make_structured_dict(record: logging.LogRecord) -> dict:
        """Creates dictionary that we want to JSON-ify.

        :param record: the LogRecord object.
        :return: dict
        """
        # next slide
```

Code

```
@staticmethod
def make_structured_dict(record: logging.LogRecord) -> dict:
    """Creates dictionary that we want to JSON-ify.

    :param record: the LogRecord object.
    :return: dict
    """
    message_str = record.getMessage()  # this interpolates '%' markers in the message.
    if isinstance(record.args, dict):  # handles case for backwards compatibility of people passing multiple args.
        user_payload = record.args
    else:
        user_payload = {'value': repr(record.args)}
    return {
        'payload': user_payload,
        'message': message_str,
        'version': '0.0.1',  # always good to have a schema version field
        'meta': {
            'level': record.levelname,
            'name': record.name,
            'pid': record.process,
            'code': {
                'file_name': record.filename,
                'file_url': record.pathname,
                # etc ...
            },
        }
    }
```

Hooking this into logger config

```
[formatters]
keys=PyBayFormatter

[formatter_PyBayFormatter]
class=package.PyBayFormatter

[handlers]
keys=console_handler

[handler_console_handler]
class=StreamHandler
formatter=PyBayFormatter
args=(sys.stdout,)

[loggers]
keys=root

[logger_root]
level=INFO
handlers=console_handler
```

```
import logging
import logging.config
import sys

config_path = 'some_dir/config.ini'
logging.config.fileConfig(config_path)

logger = logging.getLogger(__name__)
logger.info('Winning', {'key1': 1, 'key2': 'a'})

### ------- OR manually ----------------
logger = logging.getLogger(__name__)
sh = logging.StreamHandler(sys.stdout)
sh.setFormatter(PyBayFormatter())
logger.addHandler(sh)
logger.setLevel(logging.INFO)
logger.info('Hi this is a message', {'and': 'some', 'structured': 'logs'})
```

Hooking this into logger config

```
[formatters]
keys=PyBayFormatter

[formatter_PyBayFormatter]
class=package.PyBayFormatter

[handlers]
keys=console_handler

[handler_console_handler]
class=StreamHandler
formatter=PyBayFormatter
args=(sys.stdout,)

[loggers]
keys=root

[logger_root]
level=INFO
handlers=console_handler
```

```
import logging
import logging.config
import sys

config_path = 'some_dir/config.ini'
logging.config.fileConfig(config_path)

logger = logging.getLogger(__name__)
logger.info('Winning', {'key1': 1, 'key2': 'a'})

### ------- OR manually ----------------
logger = logging.getLogger(__name__)
sh = logging.StreamHandler(sys.stdout)
sh.setFormatter(PyBayFormatter())
logger.addHandler(sh)
logger.setLevel(logging.INFO)
logger.info('Hi this is a message', {'and': 'some', 'structured': 'logs'})
```

Hooking this into logger config

```
[formatters]
keys=PyBayFormatter

[formatter_PyBayFormatter]
class=package.PyBayFormatter

[handlers]
keys=console_handler

[handler_console_handler]
class=StreamHandler
formatter=PyBayFormatter
args=(sys.stdout,)

[loggers]
keys=root

[logger_root]
level=INFO
handlers=console_handler
```

```
import logging
import logging.config
import sys

config_path = 'some_dir/config.ini'
logging.config.fileConfig(config_path)

logger = logging.getLogger(__name__)
logger.info('Winning', {'key1': 1, 'key2': 'a'})

### ------- OR manually ----------------
logger = logging.getLogger(__name__)
sh = logging.StreamHandler(sys.stdout)
sh.setFormatter(PyBayFormatter())
logger.addHandler(sh)
logger.setLevel(logging.INFO)
logger.info('Hi this is a message', {'and': 'some', 'structured': 'logs'})
```

You now just need to push the logs...

Apps ⟶ Structured Logs ✔️

Structured Logs ⟶ need to go somewhere else to be used!

Many options:

- Use a specific handler to do this for you
  - e.g. SyslogHandler, FileHandler, HTTPHandler, etc.
- Flume
- Logstash
- Rsyslog
- etc.

Outline:
What is Stitch Fix/Who am I?
What *logs* am I talking about?
What are *structured logs*?
Why are *structured logs* more fun?
Caveats with *structured logs*
How to implement a *structured logger*
> Conclusion

Conclusion

- what are *structured logs*?

Structured logs allow developer intent to more easily pass through to the logs, in a more structured machine readable format.

```
logger.info('Computing %(item_id)s for client_id: %(client_id)s shipment_id: %(shipment_id)s',
            {'client_id': client_id, 'item_id': item_id, 'shipment_id': shipment_id})
```

Conclusion

- what are *structured logs*?
- why *structured logs* are more fun/leverageable?

They make developers happier because they're easier and faster to use.

They create leverage by:

- integrating more easily with other infrastructure
- reducing/removing the need for writing/maintaining regular expressions and custom parsers/analyzers.
- giving developers an incentivized single way to log.

Conclusion

- what are *structured logs*?
- why *structured logs* are more fun/leverageable?
- caveats with *structured logs*

You need to watch out for:

- typing
- naming
- serialization
- human readability

Conclusion

- what are *structured logs*?
- why *structured logs* are more fun/leverageable?
- caveats with *structured logs*
- example code to create a *JSON structured logger*

We created a custom formatter to output a JSON structured log & showed you how to hook it up.

Conclusion

- what are *structured logs*?
- why *structured logs* are more fun/leverageable?
- caveats with *structured logs*
- example code to create a *JSON structured logger*
- you learnt more about the python logging module

You can read more here:

https://docs.python.org/3/howto/logging.html
https://docs.python.org/3/howto/logging-cookbook.html

Thank you! We're hiring! Questions?

@stefkrawczyk
linkedin.com/in/skrawczyk

Try out Stitch Fix → goo.gl/Q3tCQ3