

Django Logging Demystified

DjangoCon US 2022

Supporting resources





github.com/leetrout/djangocon-us-2022



@thecodewritesme

 github.com/leetrout

Python & Go
Infrastructure, Tooling, Automation

Let's chat logging

- What is logging
- Django / Python logging setup
- Structured logging

"DIET COKE AND MENTOS
THING"? WHAT'S THAT?

OH MAN! COME ON, WE'RE
GOING TO THE GROCERY STORE.

WHY?

YOU'RE ONE OF
TODAY'S LUCKY
10,000.



Logging in Django

It's Just Python™

- Out of the box: standard Python logging facilities
- Configured via settings module
 - "dictConfig"

What is logging?

Recording Information

- Events
- Insights / data points
- Unexpected behavior

Why do we log?

✨ Bonus: Who are logs for?

Understanding

- Know what is happening
 - Debug problems
 - Monitor execution
- Communication

Context is king

- Written by people (with context):
 - Familiar with the code
- For people (lacking context):
 - Reading the code
 - Getting paged at 2am

Observability

(Part of) Observability

- Common concept: Three Pillars
 - Logs
 - Metrics
 - Traces



Opinion



- Logging can get you most of what you need
- Use performance monitoring services for **metrics** and **traces** (Sentry, HoneyComb, etc)

What is logging?

Logging is a
collection of
events...

... for ourselves and
our teammates ...

... to understand the
state of a running
system.

Logging is a feature

Logging is a feature

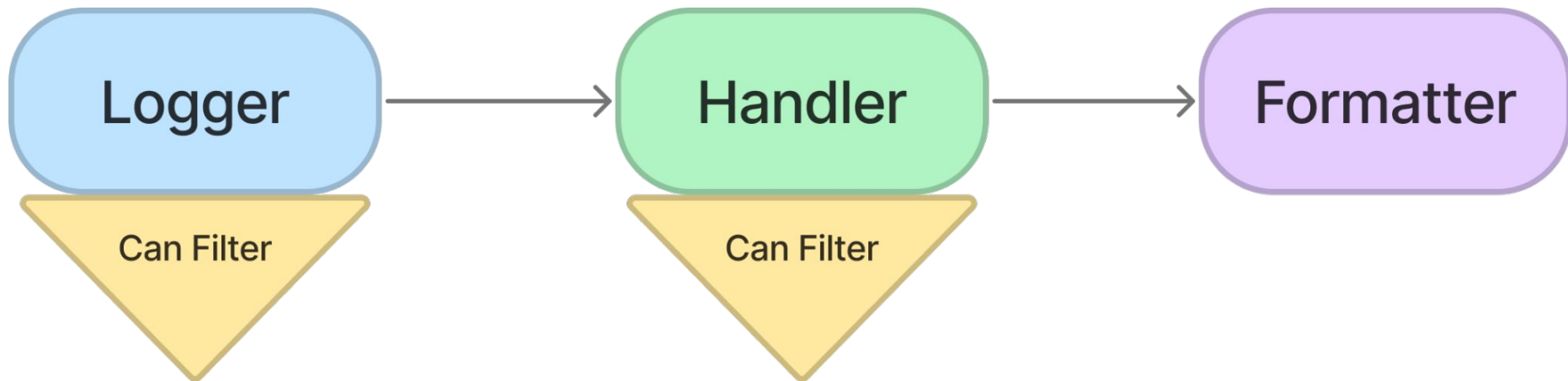
Observability is a capability

Let's chat logging

- ~~What is logging~~
- Django / Python logging setup
- Structured logging

Logging in Python

Logging component behavior



Python logging components

- Records
- Loggers
- Handlers
- Filters
- Formatters
- Levels

Levels

- (10) DEBUG
- (20) INFO
- (30) WARNING
- (40) ERROR
- (50) CRITICAL

- (0) NOTSET
- (30) WARN
- (50) FATAL

`logging._nameToLevel`

Loggers

```
>>> logger = logging.getLogger("foo")
```

```
<Logger foo (WARNING)>
```

```
>>> logger = logging.getLogger()
```

```
<RootLogger root (INFO)>
```

Loggers

```
logger.debug("a debug message")  
logger.info("an info message")  
logger.warning("a warning message")  
logger.error("an error message")
```

Quick Tip

```
logger.info(  
    "order with %d items", len(order.items)  
)  
  
logger.info(  
    "pizza with toppings: %s", toppings)  
)
```

```
logger.info(  
    f"order with {len(order.items)} items"  
)  
logger.info(  
    "order with %d items", len(order.items)  
)
```

```
<LogRecord: ..., "order with 2 items">
```

```
<LogRecord: ..., "order with %d items">
```

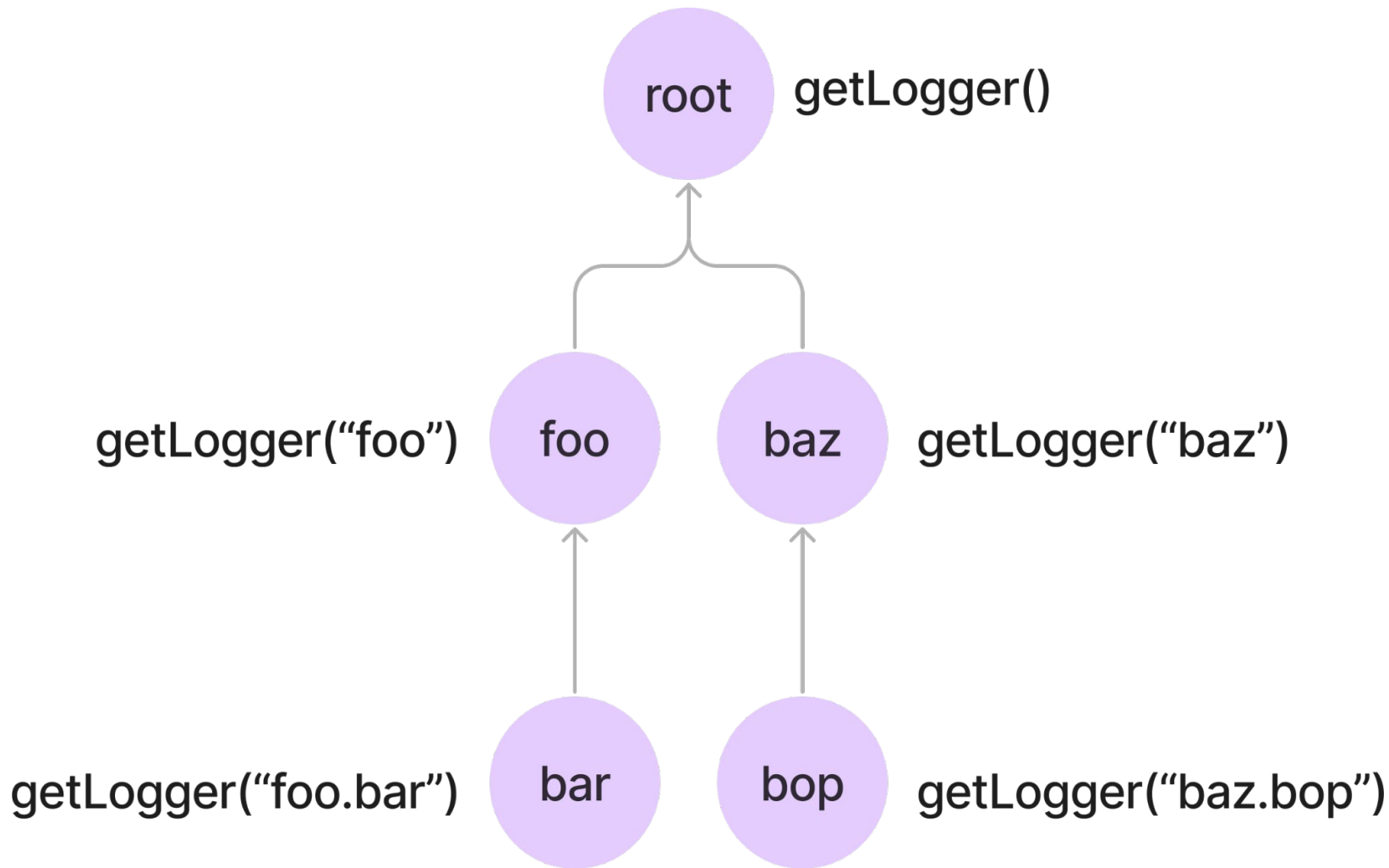
```
>>> record.msg
```

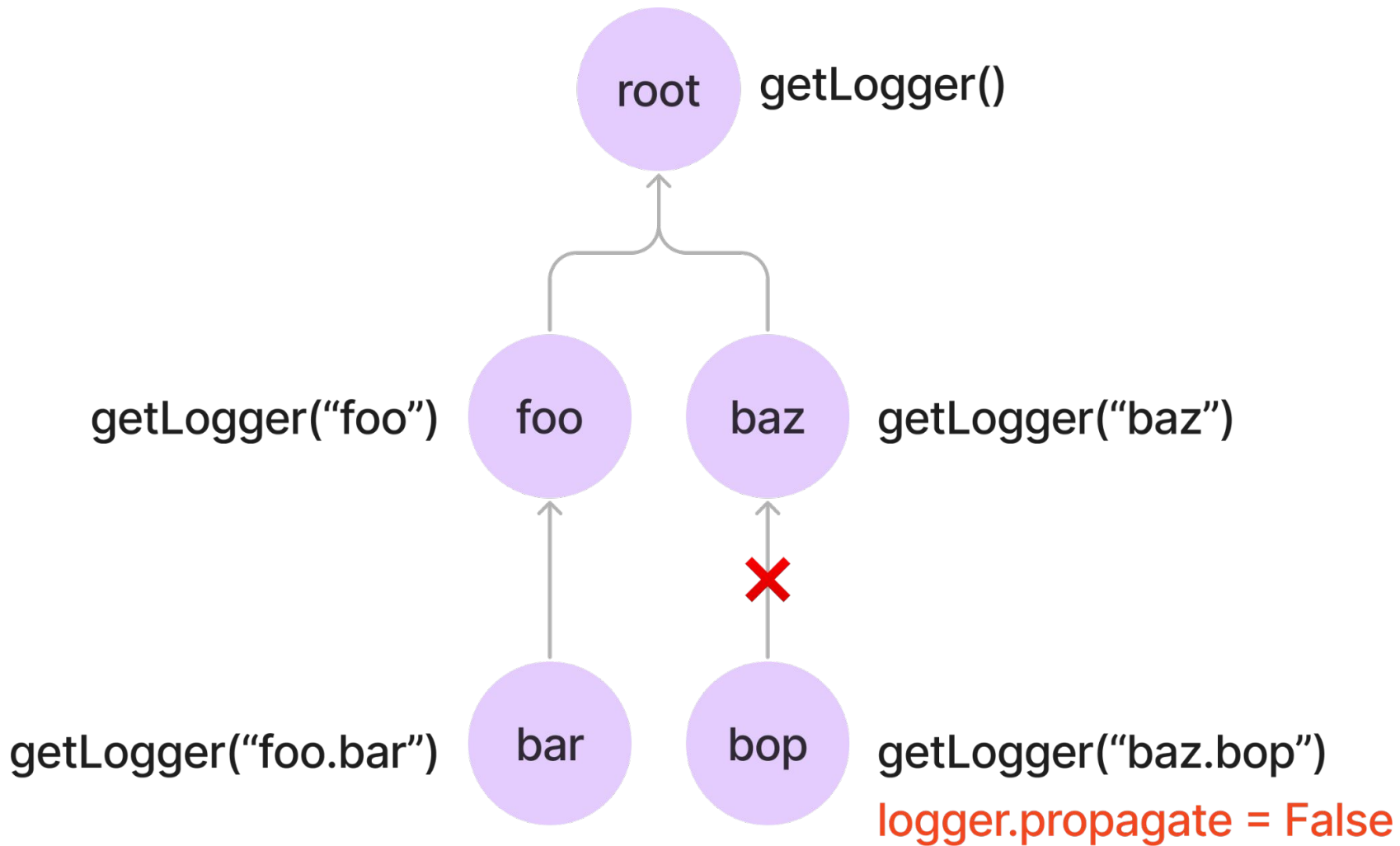
```
"order with %d items"
```

```
>>> record.args
```

```
(2,)
```

Loggers are
hierarchical





```
>>> import logging
>>> logging.getLogger("foo")
<Logger foo (WARNING)>
```

```
>>> import logging
>>> logging.getLogger().setLevel(20)
>>> logging.getLogger("foo")
<Logger foo (INFO)>
```

LogRecord

```
>>> print(some_log_record)
```

```
<LogRecord: foo, 20, .../logging_basic.py, 22, "hello!">
```

Log record attributes
are used to format
your log output

Handler

```
>>> handler = logging.StreamHandler()  
>>> handler.setFormatter(...)  
<StreamHandler <stderr> (NOTSET)>
```

Handlers can block
your program

Filters

```
class MyFilter(logging.Filter):  
    def filter(self, record) -> bool:  
        ...  
        return should_be_logged
```

Filters can be added
to loggers or handlers

Formatters

```
class ServerFormatter(logging.Formatter):  
    def format(self, record):  
        ...  
        if self.uses_server_time() ...:  
            record.server_time = self.formatTime(...)  
        return super().format(record)
```



Separation of concerns

- Code emitting logs
 - Levels and filters
- Code handling logs
 - Levels, filters and formatters

Configuring Logging

[OVERVIEW](#)[DOWNLOAD](#)[DOCUMENTATION](#)[NEWS](#)[COMMUNITY](#)[CODE](#)[ISSUES](#)[ABOUT](#)[♥ DONATE](#)

Documentation



How to configure and use logging



See also

- [Django logging reference](#)
- [Django logging overview](#)

Support Django!



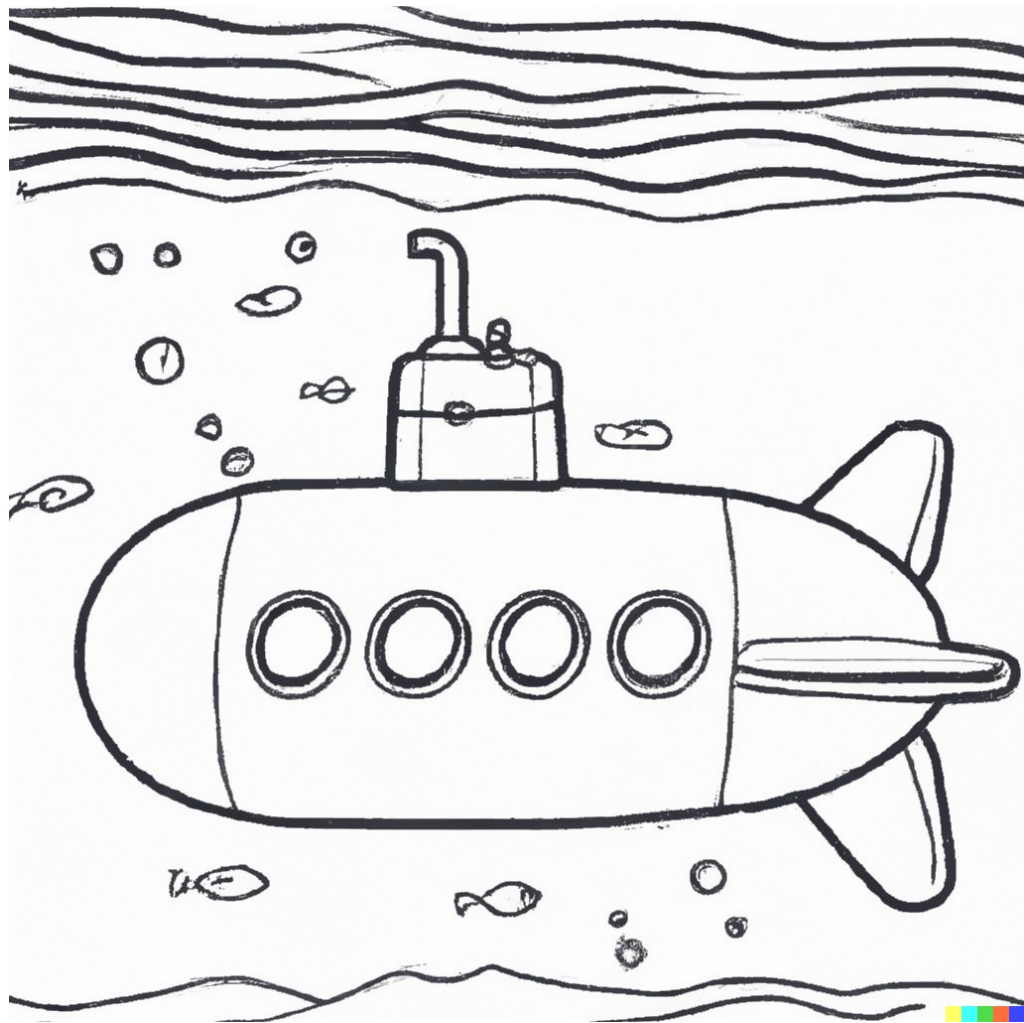
Jas.nl donated to the
Django Software
Foundation to support

Django Docs

- Logging overview
- How to configure logging
- Logging API reference

First Steps

Django gives you
reasonable defaults



dictConfig()

- Defined schema corresponding to logging components
- `logging.config.dictConfig({...})`

```
{  
  '()' : 'my.package.customFormatterFactory',  
  'bar' : 'baz',  
  'spam' : 99.9,  
  'answer' : 42  
}
```

```
my.package.customFormatterFactory(  
    bar='baz',  
    spam=99.9,  
    answer=42  
)
```

Django's config

Default behavior

- In production (debug False) errors are emailed to admins
- In development INFO level logs are reported in your console


```
{  
  "version": 1, # Required, only accepted value  
  "disable_existing_loggers": False, # Changed  
  "formatters": {...},  
  "filters": {...},  
  "handlers": {...},  
  "loggers": {...},  
}
```

```
{"version": 1, # Required, only accepted value  
"disable_existing_loggers": False, # Changed  
"formatters": {...},  
"filters": {...},  
"handlers": {...},  
"loggers": {...},  
}
```

...

```
'formatters': {  
    'django.server': {  
        '()': 'django.utils.log.ServerFormatter',  
        'format': '[{server_time}] {message}',  
        'style': '{',  
    }  
},
```

...

```
[17/Oct/2022 21:27:56] "GET / HTTP/1.1" 200 10166
```

```
{"version": 1, # Required, only accepted value
"disable_existing_loggers": False, # Changed
"formatters": {...},
"filters": {...},
"handlers": {...},
"loggers": {...},
}
```

...

```
'filters': {  
    'require_debug_false': {  
        '()': 'django.utils.log.RequireDebugFalse',  
    },  
    'require_debug_true': {  
        '()': 'django.utils.log.RequireDebugTrue',  
    },  
},
```

...

```
{"version": 1, # Required, only accepted value
"disable_existing_loggers": False, # Changed
"formatters": {...},
"filters": {...},
"handlers": {...},
"loggers": {...},
}
```

```
...
'handlers': {
    'console': {
        'level': 'INFO',
        'filters': ['require_debug_true'],
        'class': 'logging.StreamHandler',
    },
    'django.server': {...},
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
...
```



```
{"version": 1, # Required, only accepted value  
"disable_existing_loggers": False, # Changed  
"formatters": {...},  
"filters": {...},  
"handlers": {...},  
"loggers": {...},  
}
```

...

```
'loggers': {  
    'django': {  
        'handlers': ['console', 'mail_admins'],  
        'level': 'INFO',  
    },  
    'django.server': {  
        'handlers': ['django.server'],  
        'level': 'INFO',  
        'propagate': False,  
    },  
}
```

...

Logger keys are
logger names

...

```
'loggers': {
```

```
    'django': { ... }
```

...

...

```
logging.getLogger("django")
```

...

Quick Tip

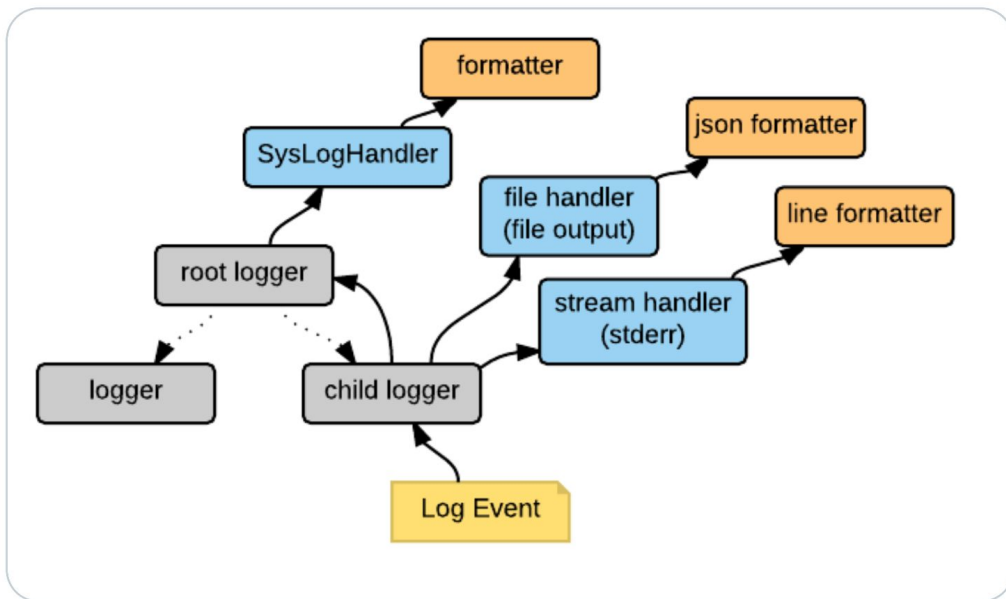
```
"django.db.backends": {  
    "handlers": ["console"],  
    "level": "DEBUG",  
    "propagate": False,  
},
```

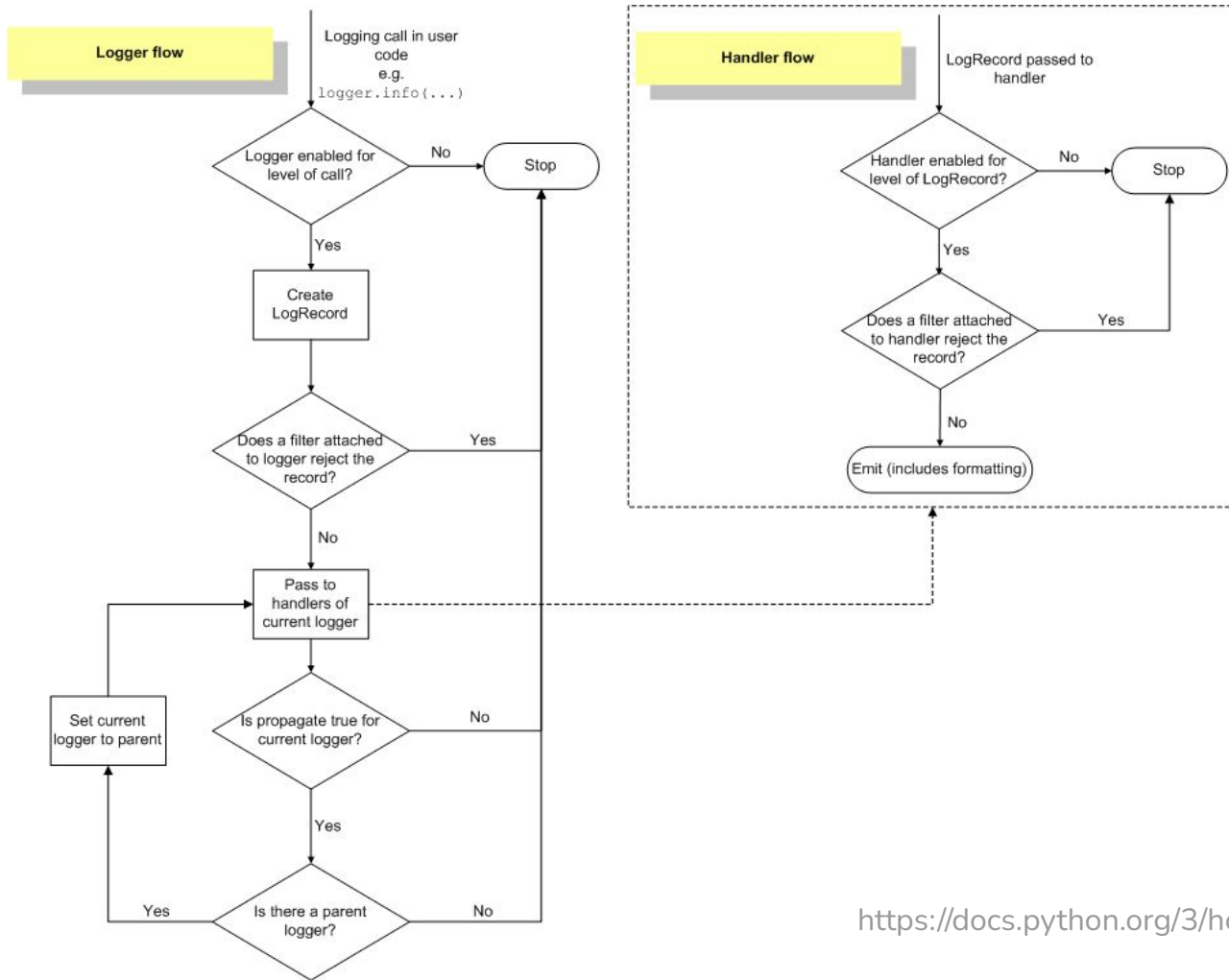
Troubleshooting logging



Jonan Scheffler 🐙
@thejonanshow

The secret to logging in Python is to give up forever and become a farmer.





print()

raise Exception()

breakpoint()

```
import pdb; pdb.set_trace()
```

Incremental logging
config changes



```
.
├── manage.py
├── pizzapro
│   ├── apps
│   │   └── core
│   │       └── views.py
│   ├── libs
│   │   └── logs.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
```

```
import logging
...
logger = logging.getLogger(__name__)
...
def place_order(request: HttpRequest) -> HttpResponse:
    ...
    logger.info(
        "order with %d items", len(order.items)
    )
```


...

```
"loggers": {  
  "pizzapro": {  
    "handlers": ["console"],  
    "level": "INFO"  
  },  

```

...

...

```
"pizzapro": {  
    "handlers": ["console"], "level": "INFO"  
},  
"pizzapro.apps": {  
    "handlers": ["console"], "level": "INFO"  
},
```

Creating loggers

- Name your logger
 - Use `__name__`
 - Don't use `__name__` in scripts or entry points which will be `"__main__"`
- Loggers exist in a hierarchy based on "dot path"

Let's chat logging

- ~~What is logging~~
- ~~Django / Python logging setup~~
- Structured logging

Structured Logging

```
logger.info("pizza order for 2 large pizzas with  
cheese and pepperoni")
```

```
logger.info("order.received", extra={
    "items": [{
        "kind": "pizza",
        "size": "large",
        "toppings": [
            "cheese",
            "pepperoni"
        ]},
        ...
    ]
})
```

Benefits

- Concise events
- Explicit context
- Easier to transform
- Compromise: good for machines & humans

Watch out for

- Sensitive data
- Too many unique attributes
- Large payloads

How to see your data

- Extra data is added as attributes to the log record
- Collect the properties by diffing a standard record with your current record in a formatter

Default attributes

```
set(  
  dir(  
    logging.makeLogRecord({})  
  )  
)
```

Extra Formatter

```
class ExtraFormatter(logging.Formatter):  
    def format(self, record):  
        these_attrs = set(dir(record))  
        extra_attrs = these_attrs - DEFAULT_RECORD_ATTRS  
        record.extra = {a: getattr(record, a) for a in extra_attrs}  
        record.msg += f" :: {record.extra}"  
        return super().format(record)
```

Before

`order.received`

`order.items_validated`

`order.placed`

After

```
order.received :: {'user_id': 'bf6d'}  
order.items_validated :: {'items': [{'kind': 'pizza'...}]  
order.placed :: {'order_id': '655f', 'user_id': 'bf6d'}
```

Advice on structuring your logs

Define your events

<subject>.<behavior>

order.received

Classify behaviors

*.<state|suffix>

<state|prefix>.*

order.items.progress

*.progress

- total_count
- current_count

```
logger.info("order.progress", extra={  
    "id": "abc123",  
    "current_count": 2,  
    "total_count": 2,  
})
```


order.duration

*.duration

- duration_ms
- start_time
- end_time

```
logger.info("order.duration", extra={  
    "id": "abc123",  
    "duration_ms": 928,  
    "start_time": '2022-09-13T20:31:06.612507',  
    "end_time": '2022-09-13T20:31:06.612507',  
})
```

Loggers own the
context

Bind data to loggers

- `logging.LoggerAdapter`
- Third party
 - `structlog.BoundLogger`
- Write your own

```
logger = logging.getLogger(__name__)

...

bound_logger = MyBindingLogger(
    logger,
    {"job_id": job_id}
)

bound_logger.info("job.started")
```

Pass logger methods

```
with log_duration(bound_logger.info, "X.duration"):  
    ...
```

Pass logger methods

```
@contextmanager  
def log_duration(logger_fn, event):  
    ...  
    logger_fn(event, extra={...})
```


Test your logs

```
with LogCapture(attributes=extract) as log:
    logger = getLogger()
    logger.debug('a debug message')
    logger.error('an error')

log.check(
    {'level': 'DEBUG', 'message': 'a debug message'},
    {'level': 'ERROR', 'message': 'an error'},
)
```

Type your log
context

TypedDict

```
class DurationLogContext(TypedDict):  
    start_time: str  
    end_time: str  
    duration_ms: int
```

Data Class

```
@dataclass
class ProgressLogContext:
    total_count: int
    current_count: int
```

Helper functions

```
def log_duration(logger_fn, event: str, extra:
DurationLogContext):
    logger_fn(event, extra=extra)
```

```
def log_progress(logger_fn, event: str, extra:
ProgressLogContext):
    logger_fn(event, extra=asdict(extra))
```

Using your logs

 ☐ **OrderException** /order/error
unable to process order

New Issue



PIZZAPRO-6






Unhandled



27s ago | 26s old

Breadcrumbs

Filter By ▾

TYPE	CATEGORY	DESCRIPTION	LEVEL	⌕ TIME
	pizzapro.libs.middleware	hello	Info	04:55:18
	pizzapro.appservice.views	<div>order.started<div><pre>{ order_id: fc9920d162ed4e19991a414327f58f0f }</pre></div></div>	Info	04:55:18
	exception	OrderException: unable to process order	Error	04:55:18

<u>VISUALIZE</u>	<u>WHERE</u>	<u>GROUP BY</u>	...
MAX(current_count)	message = order.progress	order_id	
MAX(total_count)			
<u>ORDER BY</u>	<u>LIMIT</u>	<u>HAVING</u>	
MAX(current_count) desc	None	None; include all results	

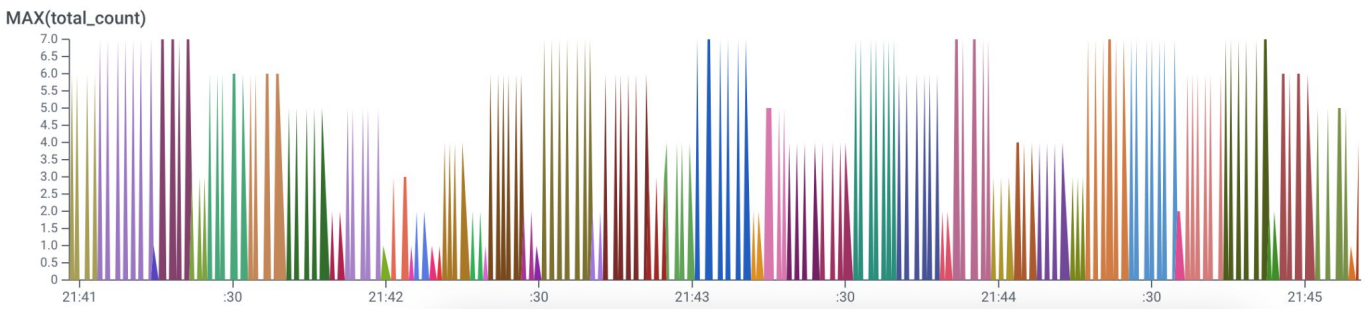
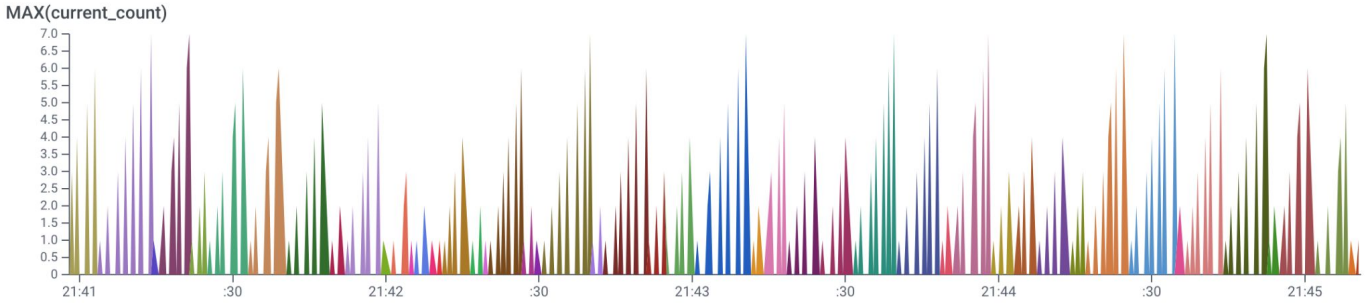
Run Query

Run a full query in seconds

Results BubbleUp Traces Raw Data

☐ Compare to Previous time range Graph Settings

Oct 16 2022, 9:40:58 PM – Oct 16 2022, 9:45:11 PM (Granularity: 500 ms)



VISUALIZE

MAX(current_count)

MAX(total_count)

WHERE

message = order.progress

GROUP BY

order_id

ORDER BY

MAX(current_count) desc

LIMIT

None

HAVING

None; include all

Let's chat logs

- ~~What is logging~~
- ~~Django / Python logging setup~~
- ~~Structured logging~~

The best logs are the
logs you have!

Thank you!



github.com/leetrout/djangocon-us-2022

BONUS SLIDES!



github.com/leetrout/djangocon-us-2022

Etymology...

Logs / logging:

Comes from *logbooks* which were used to record events.



National Maritime Museum, Greenwich, London
<https://www.rmg.co.uk/collections/objects/rmgc-object-42932>

