Diving Into Flask

Head On

Andrii V. Mishkovskyi

contact@mishkovskyi.net

Section 1

Warming up

Presentation theme

- Share our experience with Flask
- Explore inner workings of libraries we used
- Understand why things break

Why Flask?

- Well-documented
- Great API
- Easily extendable
- Well-suited for web APIs

Why Flask?

Yes, we also considered Django, Pyramid and many more

Section 2

Exploring Flask

Where we all start

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

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

Where we all start

```
@app.route('/')
def hello_world():
    return 'Hello World!'
```

Where some of us end up

```
@app.route('/albums/<int:album_id>'
           '/photos/<int:photo_id>/'
           '<string(length=4):action>')
def photo_action(album_id, photo_id, action):
    ...
```

Manual dispatch

```
@app.route('/foo',
           methods=['GET', 'POST', 'PUT'])
def foo():
    if request.method == 'GET':
        return get_foo()
    elif request.method == 'POST':
        return create_foo()
    else:
        retturn update_foo()
```

Let Flask do all the hard work

```
@app.route('/foo', methods=['GET'])
def get_foo():
    ...

@app.route('/foo', methods=['POST'])
def create_foo():
    ...

@app.route('/foo', methods=['PUT'])
def update_foo():
    ...
```

Class views with manual dispatch

```
class Foo(View):

    def dispatch_request(self):
        if request.method == 'GET':
            return self.get()
        elif request.method == 'POST':
            return self.create()
        elif request.method == 'PUT':
            return self.update()

app.add_url_rule(
    '/foo', view_func=Foo.as_view('foo'))
```

Class views with HTTP method-based dispatch

```
class Foo(MethodView):

    def get(self):
        ...

    def post(self):
        ...

    def put(self):
        ...

app.add_url_rule(
    '/foo', view_func=Foo.as_view('foo'))
```

`Flask.route`

- Decorator that calls `Flask.add_url_rule`
- `Flask.add_url_rule` creates `werkzeug.routing.Rule` and adds it to `werkzeug.routing.Map`
- `werkzeug.routing.Map` does the URL matching magic

Class views

- Can't use Flask.route decorator
- Explicitly call `Flask.add_url_rule`
- `as_view` method with creates the actual view function

Class views

```
class View(object):

    @classmethod
    def as_view(cls, name,
                *class_args, **class_kwargs):
        def view(*args, **kwargs):
            self = view.view_class(
                *class_args, **class_kwargs)
            return self.dispatch_request(
                *args, **kwargs)
        view.view_class = cls
        view.__name__ = name
        view.__doc__ = cls.__doc__
        view.__module__ = cls.__module__
        view.methods = cls.methods
        return view
```

URL matching and decomposition

- `Rule` creates regexp and collects proper converters
- `Map` holds all rules and builds the string for Rule to match
- `Converter`s convert the path parts into Python objects

URL matching and decomposition

```
>>> from werkzeug.routing import Map, Rule
>>> rule = Rule('/yada/daba/'
                '<string(length=2):bar>'
                '/<int:baz>')
>>> Map([rule])
>>> print(rule._regex.pattern)
^\|\/yada\/daba\/(?P<bar>[^/]{2})\/(?P<baz>\d+)$
>>> rule._converters
{'baz': <werkzeug.routing.IntegerConverter>,
 'bar': <werkzeug.routing.UnicodeConverter>}
>>> rule._trace
[(False, '|'), (False, '/yada/daba/'),
 (True, 'bar'), (False, '/'), (True, 'baz')]
>>> rule._weights
[(0, -4), (0, -4), (1, 100), (1, 50)]
```

URL matching and decomposition

`Rule` objects are stored in `Map` in sorted order.

```
class Rule(RuleFactory):

    def match_compare_key(self):
        return (bool(self.arguments),
                -len(self._weights),
                self._weights)

# Somewhere in Map implementation

self._rules.sort(
    key=lambda x: x.match_compare_key())
```

Modular Flask

- More manageable
- No more interference with other's work
- Pluggable views
- Turnkey functionality implementations

Introducing blueprints

- We needed API versioning
- Instant win: `url_prefix`
- Also splitting admin and API endpoints
- Ability to define per-blueprint template folder

How blueprints work

- Basically a proxy object
- That tracks if it was registered before
- The only interesting details is URL registration

How blueprints work

```
from flask import Blueprint

API = Blueprint(
    'API', __name__, url_prefix='/api/v1')

@API.route('/foo')
def foo():
    ...
```

How blueprints work

```
class Blueprint(_PackageBoundObjects):

    def record(self, func):
        ...
        self.deferred_functions.append(func)


    def add_url_rule(self, rule, endpoint=None,
                     view_func=None, **options):
        ...
        self.record(lambda s:
            s.add_url_rule(rule, endpoint,
                           view_func, **options))
```

How blueprints work

```
class Flask(_PackageBoundObject):

    def register_blueprint(self, blueprint,
                           **options):
        ...
        blueprint.register(self, options)

class Blueprint(_PackageBoundObjects):

    def register(self, app, options):
        ...
        state = self.make_setup_state(app, options)
        for deferred in self.deferred_functions:
            deferred(state)
```

Section 3

Flask and SQLAlchemy

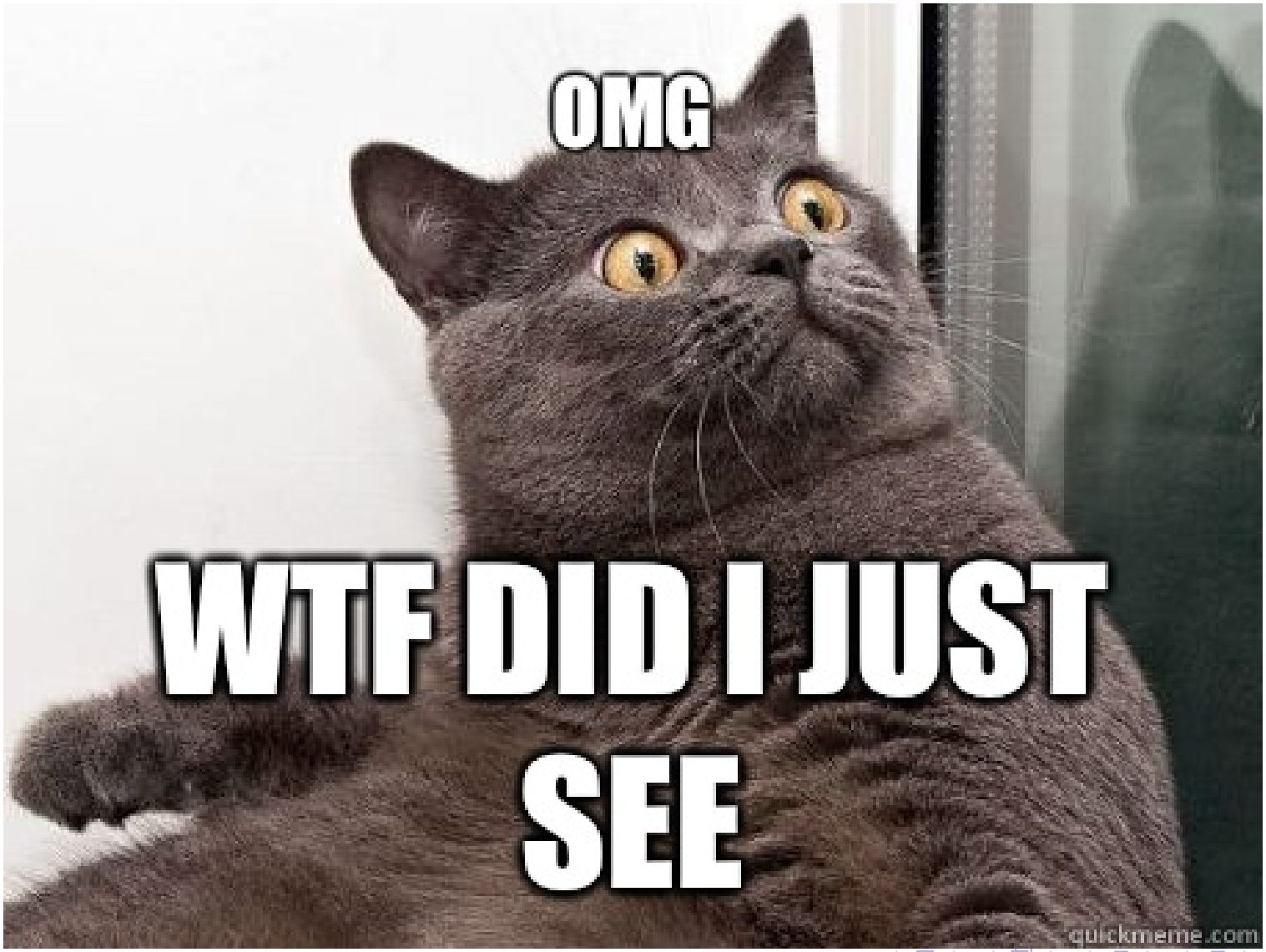
Flask-SQLAlchemy

- Full of magic
- As in, dark magic
- Say, would you guess what is the purpose of this?

Flask-SQLAlchemy

```
def _calling_context(app_path):
    frm = sys._getframe(1)
    while frm.f_back is not None:
        name = frm.f_globals.get('__name__')
        if name and \
           (name == app_path or
            name.startswith(app_path + '.')):
            funcname = frm.f_code.co_name
            return '%s:%s (%s)' % (
                frm.f_code.co_filename,
                frm.f_lineno,
                funcname
            )
        frm = frm.f_back
    return '<unknown>'
```

Flask-SQLAlchemy



SQLAlchemy and binds

- Bind is the SQLAlchemy engine or pure connection object
- Flask-SQLAlchemy gives the ability to specify bind per model
- But sometimes one model has to reference several binds

SQLAlchemy and binds

```
class AdminUsers(db.Model):

    __bind_key__ = 'admin'

    # model definition goes here
```

How Flask-SQLAlchemy does it

```
def get_bind(self, mapper, clause=None):
    if mapper is not None:
        info = getattr(
            mapper.mapped_table, 'info', {})
        bind_key = info.get('bind_key')
        if bind_key is not None:
            state = get_state(self.app)
            return state.db.get_engine(
                self.app, bind=bind_key)
    return Session.get_bind(self, mapper, clause)
```

How do we achieve master-slave support?

```
db.session.using_bind('slave').query(...)
db.session.using_bind('master').query(...)

AdminUser.query_using('admin-slave-1').all()
AdminUser.query_using('admin-slave-2').all()
```

How do we achieve master-slave support?

```
def __init__(self, *args, **kwargs):
    _SignallingSession.__init__(
        self, *args, **kwargs)
    self._name = None

def using_bind(self, name):
    self._name = name
    return self
```

How do we achieve master-slave support?

```
def get_bind(self, mapper, clause=None):
    if mapper is not None:
        info = getattr(mapper.mapped_table,
                       'info', {})
        bind_key = self._name or \
          info.get('bind_key')
    else:
        bind_key = self._name
    if bind_key is not None:
        state = get_state(self.app)
        return state.db.get_engine(
            self.app, bind=bind_key)
    else:
        return Session.get_bind(
            self, mapper, clause)
```

SQLAlchemy-migrate

- Easy to start with
- Decent documentation
- Seems abandoned
- Had to write a wrapper to run `migrate` utility

Alembic

- 7 months ago seemed to be in alpha state
- Much more mature right now
- Great documentation, great implementation
- Written by Mike Bayer himself

Section 4

Deferring your tasks

Celery features

- Removes the hassle of using amqplib/pika
- Extensive set of features
- Confusing documentation

Flask-Celery

- Flask-Script is a requirement
- Most of the commands work
- Except for starting detached celery daemons

Flask-Celery

```
from celery.platforms import detached

class CeleryDetached(celeryd):

    def run(self, **kwargs):
        sys.argv[1] = 'celeryd'
        with detached(kwargs['logfile'],
                      kwargs['pidfile']):
            os.execv(sys.argv[0], sys.argv)
```

Color formatting

Problem

Celery always colorizes logs. We don't like colors.

OH HAI COLORZ

```
(ignite)pony:ignite mishok$ ./manage.sh celeryd
/Users/mishok/.virtualenvs/ignite/lib/python2.7/site-packages/jinja2/loaders.py:214: UserW
ry/Frameworks/Python.framework/Versions/2.7/lib/python2.7/argparse.pyc, but /Users/mishok/
to sys.path
  from pkg_resources import DefaultProvider, ResourceManager, \
/Volumes/work/ignite/ignite/__init__.py:179: UserWarning: libredis could not be loaded, fa
  UserWarning)
ignite.celeryd WARNING 2012-07-02 10:56:05,594 /Users/mishok/.virtualenvs/ignite/lib/pytho

 -------------- celery@pony.local v2.5.3
---- **** -----
--- * *** * -- [Configuration]
-- * - **** ---    . broker:      amqp://guest@localhost:5672//
- ** ----------    . loader:      flask_celery.FlaskLoader
- ** ----------    . logfile:     [stderr]@WARNING
- ** ----------    . concurrency: 8
- ** ----------    . events:      ON
- *** --- * ---    . beat:        OFF
-- ******* ----
--- ***** ----- [Queues]
 --------------    . ignite-queue: exchange:ignite-exchange (direct) binding:ignite-key
```

Color formatting

Problem

Celery always colorizes logs. We don't like colors.

Solution

Add `after_setup_logger` signal that reassigns all logging formatters for Celery logging handlers.

Hijacking root logger

Problem

Root logger is hijacked by Celery's logging setup, making your logging setup useless.

Solution

Set `CELERYD_HIJACK_ROOT_LOGGER` to `False`. Or better yet, never use root logger.

Process name

Problem

Logging might brake if you want to setup logging beyond log message format. See `https://gist.github.com/721870`

> *There are three places in the code where the processName is written to a LogRecord, some of which can lead to unexpected behaviour in some scenarios*

Process name

Problem

Logging might brake if you want to setup logging beyond log message format.

Solution

Avoid those scenarios.

Keeping an eye on Celery

- Subclass `celery.events.snapshot.Polaroid`
- ???
- PROFIT

Keeping an eye on Celery

- Subclass `celery.events.snapshot.Polaroid`
- Implement `on_shutter` method
- Check various metrics
- Generate report in whatever format you need

Keeping an eye on Celery

```
from celery.events.snapshot import Polaroid

class Camera(Polaroid):

    def on_shutter(self, state):
        if not state.event_count:
            return
        print('Workers: {}'.format(
            state.workers))
        # Check state.tasks,
        # state.alive_workers,
        # etc
```

Celery + SQLAlchemy + MySQL

Problem

Each time worker starts, infamous `MySQL` error is raised:

OperationalError: (2006, 'MySQL server has gone away')

Solution

Drop the whole connection (engine) pool at worker init.

Celery + SQLAlchemy + MySQL

```
from celery import signals
from ignite.models import db

def reset_connections(**_):
    db.session.bind.dispose()

signals.worker_init.connect(reset_connections)
```

Celery + SQLAlchemy + MySQL

Problem

Session not closed if exception happens midway through transaction.

Solution

Close the session in `task_postrun` signal.

Celery + SQLAlchemy + MySQL

```
from celery import signals
from ignite.models import db

def task_postrun_handler(**_):
    try:
        db.session.commit()
    finally:
        db.session.close()

signals.task_postrun.connect(
    task_postrun_handler)
```

Celery + SQLAlchemy + MySQL

Problem

Session still not closed properly if db object loses app context. Worker hangs too if that happens.

> *RuntimeError: application not registered on db instance and no application bound to current context*

Solution

Close the session in `task_postrun` signal but only if there was an exception.

Celery + SQLAlchemy + MySQL

```
from celery import signals
from ignite.models import db

def task_postrun_handler(**_):
    try:
        db.session.commit()
    except RuntimeError:
        pass
    except Exception:
        db.session.close()

signals.task_postrun.connect(
    task_postrun_handler)
```

Section 5

Caching & profiling

Flask-Cache

- Plenty of caching decorators
- Otherwise – thin wrapper around werkzeug.contrib.cache

Really thin wrapper

```
def get(self, *args, **kwargs):
    "Proxy function for internal cache object."
    return self.cache.get(*args, **kwargs)

def set(self, *args, **kwargs):
    "Proxy function for internal cache object."
    self.cache.set(*args, **kwargs)

# Also add, delete, delete_many, etc.
```

Meh...

- Wrote our own cache classes
- With namespace support
- And consistent hashing (based on libketama)
- Also fixed and improved Python libredis wrapper

statsd

- Use `python-statsd`
- I have no more bullet points to add here
- So, there
- ...
- A picture of a cat instead!

statsd

statsd

```
def setup_statsd(app):
    host = app.config['STATSD_HOST']
    port = app.config['STATSD_PORT']
    connection = statsd.Connection(
        host=host, port=port, sample_rate=0.1)
    app.statsd = statsd.Client(
        'ignite', statsd_connection)

def statsd_metric(metric, duration=None):
    counter = app.statsd.get_client(
        class_=statsd.Counter)
    counter.increment(metric)
    if duration is not None:
        timer = current_app.statsd.get_client(
            class_=statsd.Timer)
        timer.send(metric, duration)
```

Flask-DebugToolbar

- Direct port of Django's debug toolbar
- Great at identifying bottlenecks
- We also added memory profiling (Pympler)
- Also: great example for blueprint-based plugin

Section 6

Conclusion

Flask maturity

- Flask is no longer an April Fool's joke
- Still micro, but not in terms of features
- You can and should build applications with Flask
- Flask is easy to reason about

Flask's ecosystem

- Not on par with Flask in places
- Interoperability is rough in places
- Lack's BDFL for extensions (mitsuhiko for president!)

Questions?