DEPLOYING YOUR FIRST FLASK APP - LESSONS LEARNED

BALTHAZAR ROUBEROL

@brouberol

CONTEXT

- Large web stack of several languages (Python backend & PHP frontend)
- PHP website requires complex functionalities written in Python
- Danger: maintaining two parallel codebases!
- Decision: exposing the functionalities to the frontend through a web service

CHOICE OF WEAPON: FLASK

- Essential batteries included
- A whole army of plugins
- Awesome documentation
- Very easy first steps!

⚠ FIRST STEPS ≠ PRODUCTION

LESSON #1: ALWAYS FAIL THE SAME WAY

```python
@app.route('perform/task/1', methods=['GET'])
def do_task_1():
    # do stuff
    try:
        # do sensitive stuff
    except Exception as e:
        return jsonify({'error': e.message})
    # ...
    return jsonify({'message': success_data})


@app.route('perform/task/2', methods=['GET'])
def do_task_2():
    # do stuff
    try:
        # do sensitive stuff
    except Exception:
        abort(500)
    # ...
    return jsonify({'message': success_data})
```

```
>>> r1 = requests.get('http://localhost:8000/perform/task/1')
>>> r1
<Response [200]>  # Eww, that's bad.
>>> r1.json()
{'error': 'Some failure message'}

>>> r2 = requests.get('http://localhost:8000/perform/task/2')
>>> r2
<Response [500]>
>>> r2.json()  # Now, that's bad too.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
  File "/usr/lib/python2.7/json/decoder.py", line 383, in raw_decode
    raise ValueError("No JSON object could be decoded")
ValueError: No JSON object could be decoded
```

Both are equaly bad...

LESSON #2: IT'S NOT HTTP OR JSON, IT'S BOTH

```
@app.route('perform/task/1', methods=['GET'])
def do_task_1():
    # do stuff
    try:
        # do sensitive stuff
    except Exception as e:
        return make_response(jsonify({'error': e.message}), 500)
    # ...
    return make_response(jsonify({'message': success_data}), 200)
```

`make_response` can wrap JSON data into an HTTP response.

```
>>> r1 = requests.get('http://localhost:8000/perform/task/1')
>>> r1
<Response [500]>
>>> r1.ok
False
>>> r1.json()
{'error': 'Some failure message'}

# Second time is the charm
>>> r2 = requests.get('http://localhost:8000/perform/task/1')
>>> r2
<Response [200]>
>>> r2.ok
True
>>> r2.json()
{'message': 'Some success message'}
```

More coherent behaviour.

LESSON #3: SEPARATE YOUR CONCERNS

```
@app.route('model/<_id>', methods=['GET'])
def get_model(_id):
    model_instance = Model.objects(id=_id).first()  # returns None if n
ot found
    if not model_instance:
        return make_response(jsonify({'error': 'Not found'}), 404)
    #...


@app.route('othermodel/<_id>', methods=['GET'])
def get_other_model(_id):
    model_instance = OtherModel.objects(id=_id).first()  # returns None
 if not found
    if not model_instance:
        return make_response(jsonify({'error': 'Not found'}), 404)
    #...
```

KEEP DRY

```
@app.errorhandler(mongoengine.DoesNotExist)
def not_found(error):
    """Intercepts mongoengine.DoesNotExist exceptions.
    Returns a 404 HTTP Response with a JSON error message.

    """
    return make_response(jsonify({'error': error.message}), 404)


@app.route('model/<int:_id>', methods=['GET'])
def get_model(_id):
    model_instance = Model.objects.get(id=_id)  # raises a DoesNotExist
 if not found
    #...
    return make_response(jsonify({'id': _id, 'title': model_instance.ti
tle, ...}), 200)
```

```
>>> r = requests.get('http://localhost:8000/model/0')
>>> r
<Response [404]>
>>> r.json()
{'error': 'Model matching query does not exist'}

>>> r = requests.get('http://localhost:8000/model/1')
>>> r
<Response [200]>
>>> r.json()
{'id': 1, 'title': 'Some random title'}
```

LESSON #4: LOG ALL THE THINGS

```
@app.route('perform/task/1', methods=['GET'])
def do_task_1():
    # do stuff
    try:
        # do sensitive stuff
    except Exception as e:
        app.sentry.captureException()
        return make_response(jsonify({'error': e.message}), 500)
    # ...
    return make_response(jsonify({'message': success_data}), 200)
```

Let's avoid a try/except block in all functions, now shall we?

```python
@app.errorhandler(Exception)
def log_exception(exc):
    """Intercepts all exceptions and send them to Sentry.
    Returns a 500 HTTP Response with the exception message contained
    in its JSON field.

    """
    app.sentry.captureException()
    return make_response(jsonify({'error': exc.message}), 500)


@app.route('perform/task/1', methods=['GET'])
def do_task_1():
    # do stuff
    return make_response(jsonify({'message': success_data}), 200)
```

Breakages do not always happen where expected: client termination, database error, network failure, ...

CATCH ALL THE THINGS 😎

LESSON #5: THINK IN TERM OF REQUEST CONTEXT

Famous bug "MySQL server has gone away", showing after 8 hours of inactivity

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from flask import Flask

# create SQLAlchemy global session
engine = create_engine('mysql://...')
Session = sessionmaker(bind=engine)
session = Session()

app = Flask(__name__)


@app.route('/some/route'):
def do_stuff():
    # ...
    data = session.query(...)  # will fail when MySQL kills the session
    # ...
```

USE THREAD-LOCAL SESSIONS

```
from sqlalchemy.orm import scoped_session

engine = create_engine('mysql://...', pool_recycle=7200)  # recycle ses
sion every 2 hours
Session = sessionmaker(bind=engine)

# Session is a thread-local session factory, meaning that
# calling session() will create a thread local session.
# Note that session acts like a proxy, and thus can also
# directly be used as a session. In this case, a session will
# be 'silently' created
session = scoped_session(Session)

@app.route('/some/route'):
def do_stuff():
    # ...
    data = session.query(...)  # create and use thread local session
    # ...

@app.teardown_appcontext
def shutdown_session(exception=None):
    """Remove the local session after executing the request."""
    session.remove()
```

LESSON #6: AVOID LARGE MONOLITHIC SERVER FILE

A single ~600LoC file is harder to maintain than a well separated bunch of small modules.

Me.

BLUEPRINTS

Extension to an application, providing additional functionalities.

Grouping your API calls into coherent blueprints can help improve your API design.

```python
# in myapp/blueprints/schedule.py
from flask import Blueprint

schedule = Blueprint('schedule', __name__)

@schedule.route('/format')
def format_schedule():
    # ...

@schedule.route('/format/short')
def format_short_schedule():
    # ...

@schedule.route('/format/shortest')
def format_shortest_schedule():
    # ...


# in myapp/app.py
from flask import Flask
from myapp.blueprints.schedule import schedule

app = Flask(__name__)
app.register_blueprint(schedule, url_prefix='/schedule')
# ...
```

Each blueprint can have its own static folder, template folder, test suite, etc, and can be packaged individually.

LESSON #7: DO NOT REINVENT THE WHEEL

~50 extensions registered at http://flask.pocoo.org/extensions/

- Flask-Admin: admin interface
- Flask-Bcrypt: Bcrypt support for hashing passwords
- Flask-Celery: Celery integration for Flask
- Flask-DebugToolbar: A port of the Django debug toolbar to Flask
- Flask-Login: User session management
- Flask-Mail: mail sending for Flask
- Flask-SQLAlchemy: Adds SQLAlchemy support to Flask. Quick and easy.
- Flask-Testing: Unit testing utilities for Flask.
- ...

THANK YOU!

QUESTIONS?