A PYTHON FLASK WORKSHOP
FOR FOSS4G

SPATIAL DATA APIS

SPATIAL DATA APIS

WORKSHOP GOALS

OVER THE COURSE OF THIS WORKSHOP YOU WILL:

- GET A BASIC INTRODUCTION TO SQLALCHEMY'S ORM
- USE SQLACODEGEN TO AUTOMATICALLY GENERATE A MODEL OF AN EXISTING DATABASE
- HAVE A BASIC INTRODUCTION TO THE PYTHON FLASK WEB FRAMEWORK
- USE THE FLASK-REST-APIS LIBRARY TO QUICKLY AND EASILY BUILD API ENDPOINTS
- USE THE MARSHMALLOW LIBRARY TO SERIALISE DATA AS GEOJSON
- SET UP AND USE ZAPPA TO DEPLOY YOUR PROJECT AS AN AWS LAMBDA SERVICE

There are many ways to make your data publicly available: You can upload the data to a public website (like data.gov.au from the Australian government); you can host a link to download the data from your own site; you can send people physical media in the form of a hard drive of data; you can allow people to access your database directly.

Note, this isn't going to cover the business of pointing your to your API from a custom domain. Not hard particularly, but out of scope at this point.

SPATIAL DATA APIS

PREREQUISITES

THERE ARE SOME PREREQUISITES YOU'LL NEED TO BUILD YOUR SPATIAL DATA APIS:

- PYTHON 3.6
- AN IDE OF CHOICE (I RECOMMEND PYCHARM FROM JETBRAINS)
- THE AWS CLI INSTALLED AND CONFIGURED
  - I SHOULD NOTE THAT YOU'LL NEED AN AWS ACCOUNT FOR THIS, WHICH IS FREE TO SIGN UP TO, AND WE'LL BE USING THE FREE TIER OF LAMBDA SERVICES TO DEPLOY THE APIS
- QGIS 3 TO TEST THE SERVICE

People can grab me if they absolutely don't want to set up AWS, or use an existing account, and I'll sort them out

HENRY WALSHAW

ABOUT ME

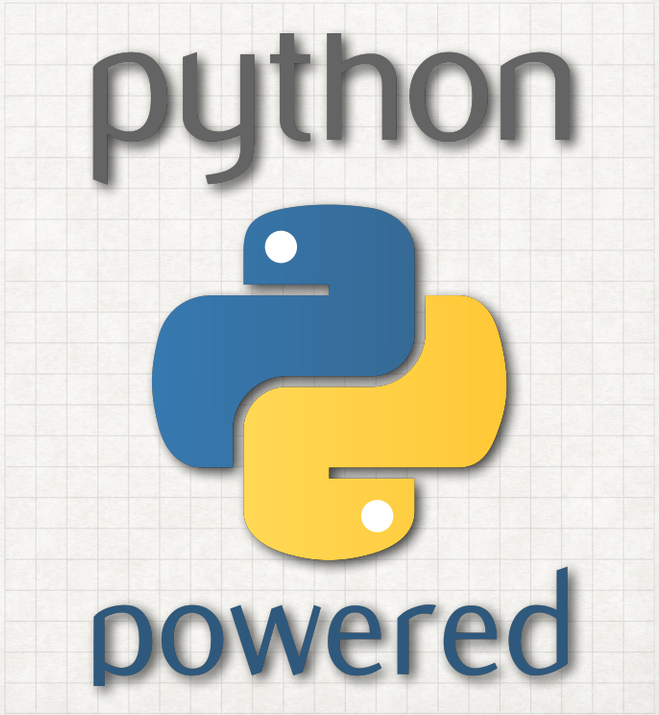
Henry has almost 15 years of experience in GIS, spatial analysis and application development, particularly in the natural resource management field. Henry's core technical expertise relates to the development and analysis of large scale spatial datasets (primarily using Python), and communicating this understanding to people including subject matter experts and the general public.

Henry has worked in government at federal and state levels, at Geoscience Australia (GA), the Victorian Government Department of Sustainability and Environment (DSE), and the Environmental Protection Agency (EPA). He has also worked in the private sector as a consultant, manager and CTO at a startup. He now teaches full time with PythonCharmers. He holds a Bachelors in Computational Science.

SPATIAL DATA APIS

WHY PYTHON?

- 4TH MOST POPULAR PROGRAMMING LANGUAGE OVERALL
  - MOST POPULAR DATA SCIENCE AND GIS LANGUAGE (BY A LONG MARGIN)
- EASY TO WRITE - YOUR TIME IS VALUABLE
- OPEN SOURCE WITH A PERMISSIVE LICENSE AND AVAILABLE ACROSS MULTIPLE SYSTEMS
- HUGE 3RD PARTY LIBRARY ECOSYTEM, INCLUDING FLASK, SQLALCHEMY AND ZAPPA



SPATIAL DATA APIS

GETTING STARTED

FIRST STEPS FIRST - CREATING A NEW PROJECT

- **I'M ASSUMING YOU'LL BE USING A TERMINAL FOR THIS, THOUGH MOST IDES WILL DO THIS FOR YOU AS WELL**
- **ONCE ITS DONE OPEN YOUR IDE OF CHOICE (I RECOMMEND PYCHARM) AND POINT IT AT THIS FOLDER**

```
$ mkdir geojson_api

$ cd geojson_api

$ git init .

$ python3 -m venv env

$ source env/bin/activate

$ pip install -U pip
```

The git init might be unnecessary, but it will make your life infinitely easier. I'd also suggest git-flow as an addition to your workflow.

Note on windows the activation command will be env\Scripts\activate

SPATIAL DATA APIS

MODELLING YOUR DATABASE

- TO BUILD AN API USING PYTHON YOU NEED TO ACCESS DATA IN PYTHON
- WE'LL BE USING THE SQLALCHEMY ORM, ALONG SIDE GEOALCHEMY 2 TO ACCESS SPATIAL DATA FROM POSTGRESQL
  - OUR PRIMARY REASON FOR FOR THIS CHOICE IS GEOMETRY SUPPORT

```
$ pip install sqlalchemy

$ pip install geoalchemy2

$ pip install psycopg2-binary

$ pip install sqlacodegen
```

Engine configuration for SQLAlchemy: http://docs.sqlalchemy.org/en/latest/core/engines.html#postgresql

Alternative ORMs include the Django ORM, PonyORM and Peewee ORM

SPATIAL DATA APIS

CONNECTING TO A DATABASE USING SQLALCHEMY

- **TO MAKE A CONNECTION YOU HAVE TO CREATE AN ENGINE AND A DRIVER**
  - **FOR POSTGRESQL THE DRIVER IS** psycopg2
- **IN THIS CASE YOU'LL BE CONNECTING TO THE**
  DATABASE: database
  HOST: host
  PORT: port
  USER: username
  PASSWORD: password

```
>>> from sqlalchemy
    import create_engine

>>> engine =
    create_engine("
      postgresql://
      username:password
      @host
      :port
      /database
    ")
```

(almost - excluding sqlite) connection strings for sqlalchemy are in the form:

database_type+driver://user:password@host:port/database

Note that this user is a trainee user so there won't be the opportunity to

SPATIAL DATA APIS

AN INTRODUCTION TO THE ORM

- AN ORM IS AN “OBJECT RELATIONAL MAPPER”
- IT ALLOWS YOU TO MODEL YOUR DATABASE TABLES AND ENTITIES AS PYTHON (CODE) OBJECTS RATHER THAN QUERYING IN SQL
- WHEN DECLARING A TABLE USING SQLALCHEMY YOU MUST INHERIT FROM A DECLARATIVE BASE

```
>>> from sqlalchemy
      .ext.declarative
    import
    declarative_base

>>> Base =
    declarative_base()
```

SPATIAL DATA APIS

DECLARING A TABLE BY HAND

- THE TABLE DECLARED AS A SUBCLASS OF THE "BASE" OBJECT IS THE EASIEST WAY TO CREATE A TABLE FROM SCRATCH
- AT A MINIMUM THE TABLE MUST HAVE A `__tablename__` AND A PRIMARY KEY
- YOU CAN DECLARE AS MANY COLUMNS AS YOU WANT
- IT'S ALSO A GOOD IDEA TO DECLARE A `__repr__` TO MAKE ERRORS EASIER

```
>>> from sqlalchemy
    import Column,
    Integer

>>> class
    VotingCentre(Base):

        __tablename__ =
            'voting_centre'

        id = Column(
            Integer,
            primary_key=True
        )
```

Primary keys should represent a primary key table in the database. They don't have to be an integer and they don't have to be the actual primary key, but it's a very good idea if they are.

SPATIAL DATA APIS

REFLECTING TABLES AUTOMATICALLY

- DECLARING A PYTHON CLASS TO REPRESENT A NEW DATABASE TABLE WORKS VERY WELL IF YOU HAVEN'T GOT A TABLE THAT ALREADY EXISTS
- IT CAN BE A LENGTHY PROCESS IF YOU HAVE A PRE-EXISTING DATABASE
- INSTEAD YOU CAN USE THE sqlacodegen COMMAND LINE TOOL TO BUILD YOUR MODELS FOR YOU

```
$ sqlacodegen
  postgresql://
    username:password
    @host
    :port
    /database
  > models.py
```

This will give you the three tables we need: District, VotingCentre and the association table. It will give you the postgis definition tables as well, which should be deleted. However it won't give you the many to many relationship which will have to be defined separately.

SPATIAL DATA APIS

GEOMETRY WITH GEOALCHEMY2

- YOU'LL NOTE THAT IN THE MODELS YOU JUST GENERATED YOU'LL HAVE GEOMETRY COLUMNS FROM GEOALCHEMY2
- GEOALCHEMY2 IS BUILT AS AN EXTENSION TO SQLALCHEMY AND SUPPORTS BOTH THE ORM AND CORE
- IF YOU'RE PLANNING ON EDITING THE GEOMETRIES DIRECTLY I'D STRONGLY ADVISE INSTALLING shapely

```
$ pip install shapely
```

Note geoalchemy 2 is currently restricted to Postresql and Postgis - you have to go back to the old (and unmaintained) geoalchemy for Oracle 11g and MS SQL Server support. Not great.

SPATIAL DATA APIS

CREATING A DATABASE SESSION

- NOW YOU'VE GOT models.py IT'S TIME TO EXPERIMENT WITH QUERYING THE DATA
- QUERYING DATA THROUGH THE ORM IS DONE VIA A SESSION
- SESSIONS ARE CREATED FROM A SESSION CLASS, WHICH IN TURN IS CREATED FROM A sessionmaker BOUND TO YOUR ENGINE

```
>>> from sqlalchemy
    import create_engine

>>> from sqlalchemy.orm
    import sessionmaker

>>> engine = create_engine(…)

>>> Session = sessionmaker(
      bind=engine)

>>> session = Session()
```

SPATIAL DATA APIS

QUERYING DATA VIA AN ORM

- YOU USE YOUR session OBJECT TO QUERY THE DATABASE
- YOU PUT QUERY OBJECTS TOGETHER BY USING STANDARD PYTHON OPERATIONS, WHICH ARE TRANSLATED TO SQL BY THE ORM

```
>>> import models

>>> query = session.query(
      models.
      VotingCentre.
      venue_name)

>>> query = query.filter(
      models.
      VotingCentre.
      venue_name.
      startswith(
        'Footscray')
      ).all()
```

Note Flask-SQLAlchemy will handle the session creation side

Queries work because the SQLAlchemy Table object overloads the standard equality etc, operators

You can also query across the many to many join (reasonably) easily:
session.query(models.VotingCentre.venue_name).join(models.VotingCentre.districts).filter(models.District.name.startswith('Footscray')).order_by(models.VotingCentre.venue_name).all()

SPATIAL DATA APIS

SPATIAL QUERIES IN SQLALCHEMY

- YOU ARE NOT RESTRICTED TO JUST QUERYING BY ATTRIBUTES - YOU CAN QUERY BY LOCATION AS WELL
- NOTE FOR OUR PURPOSES BECAUSE THE TABLE SPECIFIES AN SRID FOR THE FEATURES YOU MUST SUPPLY AN SRID FOR GEOMETRIES QUERY AGAINST

```
>>> query = session.query(
      models.District)

>>> query = query.filter(
      models.District.geom.
      ST_Intersects(
        'SRID=4326;
         POINT(144.899
         -37.798)')
    )

>>> query.first()
```

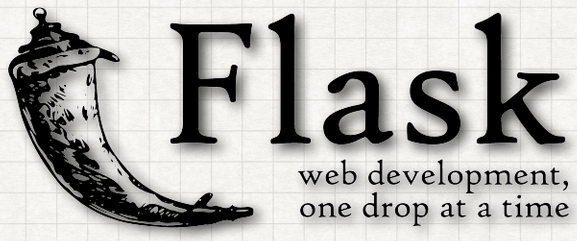
See https://geoalchemy-2.readthedocs.io/en/latest/spatial_functions.html

SPATIAL DATA APIS

A QUICK INTRODUCTION TO FLASK

- FLASK IS A MICROFRAMEWORK WRITTEN IN PYTHON
- IT IS A WSGI APPLICATION WHICH CAN BE EASILY DEPLOYED ON ANY PRODUCTION SERVER OR ON YOUR LOCAL MACHINE FOR TESTING
- FLASK IS ALSO EASY TO SETUP AS A SERVERLESS APPLICATION ENDPOINT

```
pip install -U flask
```



The idea of a micro framework is not that you have small applications. Rather the idea of the framework is that you only include the components that you require.

SPATIAL DATA APIS

A MINIMAL FLASK APPLICATION

- AT THIS POINT IT WILL MAKE YOUR LIFE MUCH EASIER TO WORK WITH AN IDE
- THE BASIC FLASK EXAMPLE DOES WORK - IT GIVES YOU A SIMPLE “HELLO WORLD” APPLICATION IN A SINGLE FILE
- WE’RE GOING TO TAKE A STEP FURTHER: IMPLEMENTING A SIMPLE APPLICATION FACTORY

```
geojson_service/
  __init__.py
  models.py
  static/
  templates/
```

PyCharm professional has support for Flask applications out of the box.

SPATIAL DATA APIS

"HELLO WORLD" APPLICATION FACTORY

- **THE APPLICATION FACTORY PATTERN IS A STANDARD FLASK DESIGN PATTERN**
- **RATHER THAN INITIALISING THE APPLICATION DIRECTLY AT THE TOP OF A PYTHON MODULE WE USE A** `create_app` **FUNCTION IN** `app.py`
  - **MAKE SURE TO IMPORT** `create_app` **INTO** `___init___.py`

```
from flask import Flask


def create_app():
    app = Flask(__name__)
    @app.route('/hello')
    def hello():
        return 'Hello, World!'
    return app
```

The application factory turns out to be pretty handy, as you can use it to make testing much easier, as well as running multiple instances of the application in the same instance.

SPATIAL DATA APIS

RUNNING THE FLASK APPLICATION

- YOU HAVE TWO CHOICES:
  - RUN FROM INSIDE THE IDE (PYCHARM PROFESSIONAL SUPPORTS THIS OUT OF THE BOX)
  - RUN THE APPLICATION FROM THE COMMAND LINE
  - THE FLASK CLI SUPPORTS BOTH create_app FACTORIES AND app DIRECT APPLICATIONS

```
$ export FLASK_ENV=
    development

$ export FLASK_APP=
    geojson_service

$ flask run
```

Test the application by going to localhost:5000/hello

Use “flask routes” to see all the endpoints in the application

SPATIAL DATA APIS

ROUTES AND RESPONSES

- AS YOU CAN SEE FROM THE MINIMAL EXAMPLE FLASK IMPLEMENTS URL ENDPOINTS (ROUTES) THROUGH PYTHON DECORATORS
- IF YOU LOOK AT THE DEVELOPMENT SITE YOU WILL SEE THE RETURNED TEXT RENDERED AS HTML (FLASK'S DEFAULT) - THE RESPONSE
- RATHER THAN SIMPLY REGISTERING ENDPOINTS, WE'LL USE BLUEPRINTS IN views.py

```
from datetime import datetime

from flask import Blueprint,
Response

base_pages = Blueprint(
    'base_pages', __name__)

@base_pages.route('/heartbeat')

def heartbeat():

    return Response(
      datetime.now()
        .isoformat(),
      mimetype='text/plain')
```

Use app.register_blueprint(base_pages) to register the blueprint

Delete the hello endpoint

SPATIAL DATA APIS

FLASK SERVER CONFIGURATION

- DEPENDING ON THE FLASK EXTENSIONS YOU ARE USING YOU WILL NEED TO PROVIDE CONFIGURATION
- AT A MINIMUM YOU SHOULD SUPPLY A "SECRET_KEY"
- NEVER COMMIT A SECRET KEY TO YOUR VERSION CONTROL REPOSITORY

```
app.config['SECRET_KEY'] =
    b'your secret key'

# or better:

app.config.from_envvar(
    'GEOJSON_SERVER_SETTINGS')

# where
# GEOJSON_SERVER_SETTINGS
# is /path/to/settings.cfg
```

Don't forget http://www.gnuterrypratchett.com/#flask

Don't commit the secret key to the repository!

python -c 'import os; print(os.urandom(16))'

Also, note that the config file is assumed to be a Python file, and any uppercased name will be added as a server configuration object.

SPATIAL DATA APIS

FLASK-SQLALCHEMY

- **AT THIS POINT YOU WANT TO GET ACCESS TO YOUR DATABASE WITHIN YOUR APPLICATION**
- **FLASK-SQLALCHEMY IS AN EXTENSION TO LINK FLASK AND SQLALCHEMY**

```
pip install flask-
sqlalchemy
```

- **ONCE FLASK_SQLALCHEMY IS INSTALLED YOU MUST REPLACE THE** Base **IN** models.py **WITH** db.Model **AND REFERENCES TO SQLALCHEMY OBJECTS WITH** db.Object

```
from flask_sqlalchemy import
SQLAlchemy

# … other imports

db = SQLAlchemy()

class District(db.Model):

# … etc
```

Must replace metadata for Table declaration with db.Model.metadata - note that db.Model is a declarative base object. The application will handle sessions by itself.

In app.py need in create_app:

```
from .models import db
db.init_app(app)
```

SPATIAL DATA APIS

FLASK-SQLALCHEMY SETUP

- **YOU WILL NEED TO MODIFY YOUR APPLICATION CONFIGURATION WITH TWO ENVIRONMENT VARIABLES:**
- SQLALCHEMY_DATABASE_URL **MUST BE THE CONNECTION STRING TO THE DATABASE:**

`postgresql+psycopg2://username:password@host:port/database`

- SQLALCHEMY_TRACK_MODIFICATIONS **SHOULD BE** False **(UNLESS YOU ARE TRACKING DATABASE INTERACTION, WHICH WE'RE NOTE)**

SPATIAL DATA APIS

QUERYING THE DATABASE WITH FLASK-SQLALCHEMY

- **WE'LL CREATE A NEW BLUEPRINT TO QUERY ALL OF THE VOTING CENTRES INSIDE A GIVEN DISTRICT;** voting_centre_views.py
- **IN** templates/ **CREATE AN** output.csv **FILE CONTAINING:**

```
{{ fieldnames|join(',') }}

{% for row in rows -%}

    {{ row|map('replace', 'NaN', '')|join(',') }}

{% endfor %}
```

```
from flask import Blueprint, Response, render_template
from .models import db, VotingCentre, District
vc_pages = Blueprint('voting_centres', __name__)
@vc_pages.route('/district/<district>')
def get_centres_in_district(district):
    query = db.session.query(VotingCentre.venue_name).join(
        VotingCentre.districts).filter(db.and_(
            VotingCentre.venue_type == 'Voting Centre',
            District.name.contains(district)
        )
).order_by(VotingCentre.venue_name)
    return Response(render_template('output.csv',
        rows=query.all(), fieldnames=['venue_name']),
        mimetype='text/csv')
```

Don't forget to add as a registered app with app.register_blueprint(vc_pages, url_prefix='/centres')

SPATIAL DATA APIS

MARSHMALLOW AND SERIALISATION

- MARSHMALLOW IS A SERIALISATION LIBRARY DESIGNED TO MAKE IT EASY TO TRANSFORM YOUR DATA FROM PYTHON OBJECTS TO OTHER FORMATS (I.E. JSON) AND VICE VERSA
- DOWNLOAD HTTP://BIT.LY/FOSS4G_MARSHMALLOW AND ADD IT TO YOUR PROJECT

```
$ pip install marshmallow

$ pip install flask-
    marshmallow

$ pip install marshmallow-
    sqlalchemy
```

Happily this doesn't require too many changes to simply work with regular SQLAlchemy, though it's not as necessary there.

You can demonstrate the schema via the flask shell if it needs demonstrating (probably not a bad place to start actually...)

Again, don't forget to add the ma.init_app(app) to the create_app (it must be **AFTER** the db initialisation).

SPATIAL DATA APIS

CREATING A SIMPLE SCHEMA

- **MARSHMALLOW DEFINES SCHEMA CLASSES THAT ARE THE REPRESENTATIONS OF COMPLEX OBJECTS**
- **IT IMPLETEMENTS** dump **AND** load **METHODS WHICH CONVERT TO AND FROM PYTHON DICTIONARIES**
  - **IN MANY WAYS THIS IS A FANCY JSON SERIALISER - ALSO SEE** dumps **AND** loads
- **THE** ma **OBJECT MAKES IT VERY EASY TO CREATE A SCHEMA FROM A SQLALCHEMY MODEL**

```
>>> from geojson_service.
      serializer_utils import ma

>>> from geojson_service.
      models import
      VotingCentre, db

>>> class Flattener(
      ma.ModelSchema):
        class Meta:
            model =
              VotingCentre

>>> Flattener().dump(
      db.query(
        VotingCentre).first())
```

Again, use flask shell for the example here (or run from within PyCharm)

SPATIAL DATA APIS

MODIFYING THE SCHEMA DUMP

- **MARSHMALLOW ALSO ALLOWS YOU TO MODIFY THE WAY A SCHEMA IS DUMPED BY IMPLEMENTING** wrap_feature **AND** wrap_envelope
  - **YOU SHOULD IMPLEMENT THE INVERSE** unwrap **FUNCTIONS ALSO**
- **THIS IS WHAT THE SAMPLE CODE HAS DONE, SO ALL THAT IS REQUIRED IS CLASS INHERITANCE**

```
>>> from geojson_service.
      serializer_utils import
MarshmallowGeoJSON

>>> class GeoJSONSchema(
      MarshmallowGeoJSON,
      ma.ModelSchema):
        class Meta:
            model =
              VotingCentre

>>> GeoJSONSchema(
      many=True).dump(
      db.query(
        VotingCentre).all())
```

SPATIAL DATA APIS

CREATING THE SERIALISERS

- YOU DON'T WANT TO HAVE TO RECREATE THE SERIALISERS FROM SCRATCH EVERY TIME YOU WANT TO USE THEM
  - ESPECIALLY GIVEN THEY DON'T CHANGE
- CREATE A serialisers.py WHICH IMPLEMENTS THE CLASSES AND CREATES CLASS INSTANCES

```
from .serializer_utils import
  MarshmallowGeoJSON, ma

from .models import VotingCentre

class VotingCentreGJ(
  MarshmallowGeoJSON,
  ma.ModelSchema):

    class Meta:

        model = VotingCentre

voting_centre_collection =
  VotingCentreGJ(many=True)

voting_centre_feature =
  VotingCentre()
```

We can if we're getting through at a superfast speed go away and create another view that returns a JSON string with dumps, but it's probably unnecessary at this point

SPATIAL DATA APIS

THE CRUD PATTERN

- CONSIDER THE VotingCentre WHICH YOU WANT TO EXPOSE TO THE USERS. YOU MIGHT CHOOSE THE URL /centres
- FLASK HAS PLUGGABLE VIEWS, BASED ON THE CLASS BASED VIEWS FROM DJANGO
- FLASK METHOD VIEWS ARE PLUGGABLE VIEWS DESIGNED TO IMPLEMENT HTTP VERBS AND ARE ESPECIALLY USEFUL FOR REST APIS

ENDPOINT	ACTION	RESULT
/centres	GET	LIST OF ALL CENTRES
/centres	POST	CREATE A NEW CENTRE
/centres/<id>	GET	GET CENTRE WITH id
/centres/<id>	PUT / PATCH	EDIT CENTRE WITH id
/centres/<id>	DELETE	DELETE CENTRE WITH id

SPATIAL DATA APIS

A FIRST REST ENDPOINT

- **YOUR FIRST STEP IN BUILDING A REST API IS TO BUILD A** MethodView
- **YOU'LL DO THIS IN A NEW** apis.py **FILE**
- **THE** MethodView **SUBCLASS NEEDS TO IMPLEMENT METHODS WITH HTTP VERB NAMES**
- **IN THIS EXAMPLE THIS MEANS WRITING A** get **METHOD**

```
from flask import jsonify

from flask.views import MethodView

from .serialisers import
  voting_centre_collection

from .models import VotingCentre

class CentresAPI(MethodView):

    """Voting centres"""

   def get(self):

        centres =
          VotingCentre.
          query.limit(10).all()

        return jsonify(
            voting_centre_collection.dump(
            centres).data)
```

flask_rest_api is actually written by one of the Marshmallow maintainers

SPATIAL DATA APIS

MAKING THE VIEW USABLE

- AT THE MOMENT THERE IS NO URL TIED TO THE VIEW
- TO ADD URLS YOU FIRST NEED TO CREATE AN INSTANCE OF THE MethodView USING THE as_view CLASS METHOD
- ADD A NEW BLUEPRINT, AND ADD THE URL REQUESTS TO THE METHOD VIEW
- ONCE THIS IS WORKING YOU HAVE A (BASIC) REST API

```
from flask import Blueprint

api_pages = Blueprint('api',
__name__)

centre_api_view =
CentresAPI.as_view('centres_ap
i')

api_pages.add_url_rule('/
centres',
view_func=centre_api_view,
methods=['GET',])
```

Don’t forget to register this blueprint!

You can use the default pattern to minimise the number of views as well:

```
user_view = UserAPI.as_view('user_api')
app.add_url_rule('/users/', defaults={'user_id': None},
                 view_func=user_view, methods=['GET',])
app.add_url_rule('/users/', view_func=user_view, methods=['POST',])
app.add_url_rule('/users/<int:user_id>', view_func=user_view,
                 methods=['GET', 'PUT', 'DELETE'])
```

SPATIAL DATA APIS

DOCUMENTING YOUR API

- YOU COULD LEAVE THE API WHERE YOU HAVE IT. BUT THE KEY TO A GOOD API IS GOOD DOCUMENTATION
- YOU COULD MANUALLY ADD OPENAPI (FORMERLY "SWAGGER") DOCS WITHOUT TOO MUCH EFFORT USING A LIBRARY LIKE flasgger
- BETTER YET, BY UPDATING YOUR settings.cfg YOU CAN USE flask-rest-apis TO DO THE HEAVY LIFTING

```
$ pip install
    flask_rest_apis
```

```
OPENAPI_URL_PREFIX='/api/doc'

OPENAPI_SWAGGER_UI_PATH=
  '/swagger'

OPENAPI_SWAGGER_UI_VERSION=
  '3.18.3'
```

flask_rest_apis is written by one of the Marshmallow maintainers

SPATIAL DATA APIS

THE FLASK-REST-APIS LIBRARY

- flask-rest-apis **IS WRITTEN BY ONE OF THE MARSHMALLOW MAINTAINERS TO MAKE IT EASY TO TURN A MARSHMALLOW SCHEMA INTO API DOCUMENTATION**
- **IT'S A BIT IMMATURE AT THE MOMENT, SO ADDING THE BLUEPRINT ISN'T AS STRAIGHTFORWARD AS I'D LIKE**
- **FIRST STEP IS TO MODIFY THE** Blueprint **AND WRITE A REGISTRATION FUNCTION**

```
from flask_rest_api import Api,
  Blueprint

api_pages = Blueprint(‘api',
    __name__,
    description='Operations of voting
      centres',
    url_prefix='/api'
)

api = Api()

def register_definitions():

    api.spec.definition(
        'VotingCentre',
        schema=VotingCentreGJ)

    api.register_blueprint(api_pages)
```

The registration function takes care of adding the schemas to the docs so they can be referenced later, and you have to add the following to your create_app:

```
from .apis import api, register_definitions
api.init_app(app)
register_definitions()
```

SPATIAL DATA APIS

UPDATING THE API ENDPOINT

- **NOW THAT YOU'VE GOT THE UPDATED BLUEPRINT SETUP YOU NEED TO UPDATE THE** MethodView:
- **IT SHOULD BE DECORATED WITH A SCHEMA**
- **IT SHOULD RETURN DATA TO BE SERIALISED**
- **RUN THE SERVER AND GO TO** http://localhost:5000/api/doc/swagger **TO SEE THE GENERATED DOCS**

```
@api_pages.route('/centres')

class CentresAPI(MethodView):

    """Voting centres"""

    @api_pages.response(
      VotingCentreGJ(many=True))

    def get(self):

        """Get 10 voting centres"""

        centres = VotingCentre.
          query.limit(10).all()

        return centres
```

Now it's just a matter of adding as much (or as little) to the API as you want.

SPATIAL DATA APIS

PREPARING FOR DEPLOYMENT

- YOU'RE FINALLY READY TO DEPLOY THE API TO AWS LAMBDA
- MAKE SURE YOU'VE GOT YOUR AWS CONFIGURATION SET UP CORRECTLY
- YOU'LL NEED A run.py FILE OUTSIDE YOUR MODULE THAT CAN INITIALISE THE app

```
from geojson_service import
  create_app

app = create_app()

if __name__ == '__main__':

    app.run()
```

SPATIAL DATA APIS

ZAPPA

- YOU'RE FINALLY READY TO DEPLOY THE API TO AWS LAMBDA
- MAKE SURE YOU'VE GOT YOUR AWS CONFIGURATION SET UP CORRECTLY
- INSTALL ZAPPA AND THEN RUN THE INIT TO WALK THROUGH THE PROJECT SETUP FOR A DEV SERVER
  - WHEN YOU'RE ASKED ABOUT THE APPLICATION FUNCTION ENTER run.app

```
$ pip install zappa

$ pip freeze >
    requirements.txt

$ zappa init
```

SPATIAL DATA APIS

ZAPPA SETTINGS

- YOU'LL NOW HAVE A zappa_settings.json WHICH CONTAINS THE SETTINGS FOR YOUR CURRENT DEPLOYMENT
- ADD THE environment_variables KEY SO YOU CAN SET THE .cfg LOCATION
- MAKE SURE THE aws_region KEY IS SET AS WELL

```
{
    "dev": {
        "app_function": "run.app",
        "aws_region": “ap-southeast-2",
        …
       "environment_variables": {
            "GEOJSON_SERVER_SETTINGS":
              "dev_settings.cfg"
        }
    }
}
```

SPATIAL DATA APIS

DEPLOYING YOUR APPLICATION

- THIS IS THE EASY BIT - RUN THE DEPLOY COMMAND FOR THE STAGE YOU WISH TO DEPLOY
- ZAPPA WILL PACKAGE THE APPLICATION AND CREATE A NEW AWS LAMBDA APPLICATION FOR YOU. IF IT WORKS YOU'LL GET A URL BACK YOU CAN TRY
- YOU CAN ALSO WATCH LOGS, UPDATE, AND UN-DEPLOY AS REQUIRED

```
$ zappa deploy dev

$ zappa tail dev

$ zappa update dev

$ zappa undeploy dev
```

Best way to test the application is via the API - test if they're available!

SPATIAL DATA APIS

THANKS FOR COMING