Python Workshop Documentation

Release 0.1

Camptocamp

July 19, 2010

CONTENTS

CHAPTER

ONE

GETTING STARTED

In this workshop you will learn about several technologies that can be used for developing web mapping applications with Python. Some of them are general-purpose web development technologies while other are geospatial-specific. Here are these technologies:

- Pylons http://pylonshq.com - *Pylons is a lightweight web framework emphasizing flexibility and rapid development*
- SQLAlchemy http://www.sqlalchemy.org - *The Python SQL Toolkit and Object Relational Mapper*
- GeoAlchemy http://geoalchemy.org - *Using SQLAlchemy with Spatial Databases*
- Shapely http://trac.gispython.org/lab/wiki/Shapely - *Shapely lets you do PostGIS-ish stuff outside the context of a database*
- geojson http://trac.gispython.org/lab/wiki/GeoJSON - *A reference implementation of the GeoJSON specifications*
- TileCache http://tilecache.org - *Web Map Tile Caching*
- MapFish http://www.mapfish.org - *MapFish is a flexible and complete framework for building rich web-mapping applications*

1.1 Start the virtual machine

For this workshop the Debian GNU/Linux operating system will be used. It will be executed in a VirtualBox virtual machine. To start the Debian operating system open VirtualBox and start the system named `python_workshop`.

Note: If there's no system named `python_workshop` you will have to create one in VirtualBox, using the `python_workshop.vdi` file as the Hard Disk.

Use `workshop` as the username and password to log into the system.

1.2 Open the documentation

This document (the one you're reading right now) is available in the Debian system in HTML form. To view it launch FireFox and open http://localhost/python_workshop.

1.3 Create a directory for the workshop

Open a terminal, and create a directory for this workshop:

```
$ mkdir python_workshop
```

This directory will be your working directory for this workshop.

1.4 Create the Python environment

The various Python packages that you will use in this workshop will be installed in a virtual Python environment.

A virtual Python environment is a Python environment isolated from the main, system-wide Python environment. A virtual Python environment allows, among other things, to install Python packages as a regular user (with no admin priviledges).

You're going to create the virtual environment in the `python_workshop` directory so start by changing to this directory:

```
$ cd python_workshop
```

You can now create the virtual environment using the `virtualenv` command. For example, to create a virtual environment named `vp` use this command:

```
$ virtualenv --no-site-packages vp
```

The `--no-site-packages` option is used to fully isolate the virtual environment from the main environment; although not mandatory this option is recommended.

The `virtualenv` command should have created a directory named `vp`, make sure this is the case.

1.5 Activate the Python environment

Now activate the virtual environment with:

```
$ source vp/bin/activate
```

Your prompt should now be changed, it should look like this:

```
(vp) $
```

With the virtual environment activated if you enter `python` the Python interpreter from the virtual environment will be executed.

1.6 Miscellaneous notes

You will use PostgreSQL in this workshop. You can use `pgAdmin` to connect to the PostgreSQL database and verify the creation of tables, rows, etc. (`pgAdmin` should be installed in the Debian image, and a button to start it is probably available in the desktop's top panel.)

You will create an application named `WorkshopApp` in this workshop, the final version of the application is available in the Debian system in `/usr/local/workshops/python_workshop/materials/WorkshopApp`.

CHAPTER

TWO

PYLONS AND SQLALCHEMY

In this module you will learn about the Pylons framework and the SQLAlchemy database toolkit. More specifically you will learn how to create a Pylons application (or project), and how to use SQLAlchemy in this application.

Note: The virtual environment must be activated prior to executing the commands provided in this module.

2.1 Install Pylons and SQLAlchemy

You are going to start by installing Pylons 1.0, and SQLAlchemy 0.6.1.

To install Pylons use:

```
(vp) $ easy_install "Pylons==1.0"
```

The `easy_install` command downloads the packages from the official Python package repository (http://pypi.python.org) and installs them in the Python environment (the virtual environment here).

You should now have Pylons installed. You can check that using this command:

```
(vp) $ paster create --list-templates
```

This command should output this:

```
Available templates:
basic_package:   A basic setuptools-enabled package
paste_deploy:    A web application deployed through paste.deploy
pylons:          Pylons application template
pylons_minimal:  Pylons minimal application template
```

Note: The `paster` command comes from `Paste` http://pythonpaste.org/. `Paste` is a low-level framework for web development, Pylons heavily relies on it.

To install SQLAlchemy use:

```
(vp) $ easy_install "SQLAlchemy==0.6.1"
```

You will use PostgreSQL as the database system in this workshop, so the PostgreSQL Python driver must be installed as well:

```
(vp) $ easy_install "psycopg2==2.0.14"
```

2.2 Create application

You can now create the Pylons application with:

```
(vp) $ paster create -t pylons WorkshopApp
```

`WorkshopApp` is the name of the Pylons application, you can pick any name of your choice , although it's assumed that you choose `WorkshopApp` in the rest of the document.

When asked what template engine to use answer `mako`, which is the default. When asked if SQLAlchemy 0.5 configuration is to be included, answer `True`, as your application will include web services relying on database tables.

Note: Although Pylons assumes SQLAlchemy 0.5 is used SQLAlchemy 0.6 and Pylons 1.0 are fully compatible.

You should now have a directory named `WorkshopApp`. This directory contains your application files, mainly Python files.

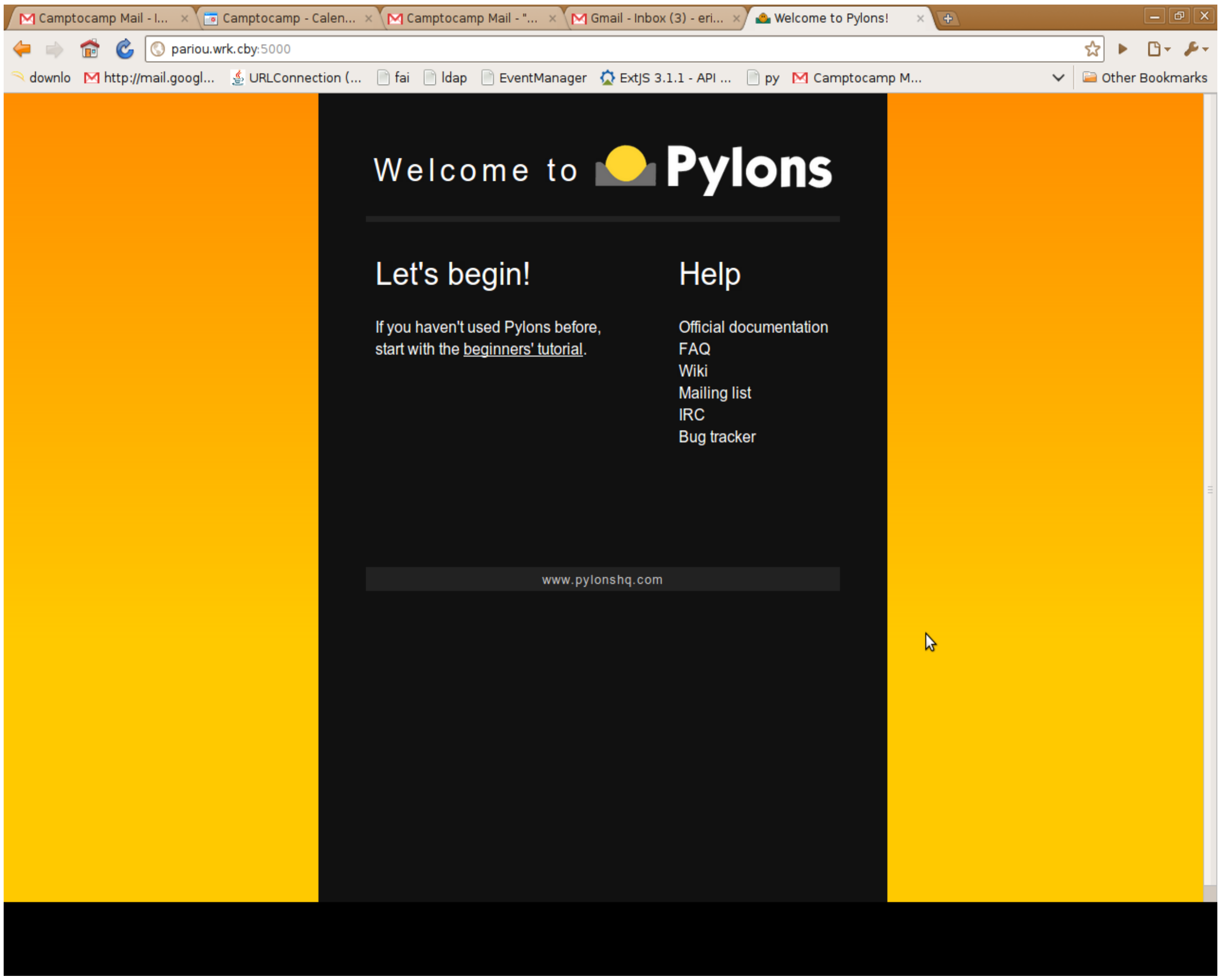
Now is the time to check that your Pylons application works. For this go into the `WorkshopApp` directory and start the application:

```
(vp) $ cd WorkshopApp
(vp) $ paster serve development.ini
```

This command starts your application in the Paste web server, which is a pure-Python web server, commonly used during development.

Note: You can use `paster serve --reload development.ini` so the Paste web server reloads itself when files are modified in the application.

Open http://localhost:5000 in your web browser, you should get the default page:



Welcome to Pylons

Let's begin!

If you haven't used Pylons before, start with the beginners' tutorial.

Help

Official documentation
FAQ
Wiki
Mailing list
IRC
Bug tracker

www.pylonshq.com

2.3 Study application

The following sub-sections give you a quick tour through the directories and files of your Pylons application. Take some time to browse those directories and files, so you get a sense of how the application is structured.

The application's main directory, `WorkshopApp`, contains:

`development.ini` This is the application's configuration file. This file includes things like the IP address and TCP port the server should listen on, the database connection string, etc.

`setup.cfg` and `setup.py` These files control various aspects of how the Pylons application is packaged when you distribute it.

`workshopapp`

This is the main application directory, its name depends on the application name given as the argument to the `paster create` command. The main sub-directories of this directory are: `controllers`, `model`, `lib`, `config`, `tests`, `templates`, and `public`.

`controllers` The `controllers` directory contains the application controllers. The controllers are the components that handle HTTP requests and send HTTP responses. They often interact with the `model` and `templates` code.

`model` The `model` directory is where the database model is configured. This is basically where tables and relations are defined.

`lib` The `lib` directory includes Python code shared by different controllers, and third-party code.

`config` The `config` directory includes Python code generated by the framework and exposed to the application for customization.

`tests` The `tests` directory is where you can add Python automated tests for the application.

`templates` The `templates` directory is where view templates are stored. Note that we won't write templates as part of this workshop, as the HTML rendering will mostly be done client side.

`public`

The `public` directory includes the application's static files, i.e. HTML, CSS, JavaScript files, etc.

2.4 Set up application

You now need to specify the location of the database in the configuration of the application. For that edit the `development.ini` file and change the value of the `sqlalchemy.url` option as follows:

```
sqlalchemy.url = postgresql://www-data:www-data@localhost:5432/python_workshop
```

This setting assumes that a PostgreSQL instance executes on the local machine, listens on port 5432, and includes a database named `python_workshop`, which user `www-data` can access using `www-data` as the password.

Note: The VirtualBox Debian image comes with a PostgreSQL server including the `python_workshop` database, and the PostgreSQL server is started automatically at boot time.

2.5 Create SQLAlchemy model

The `python_workshop` database includes a table named `summits`, which includes information about summits of France. Here you're going to define the SQLAlchemy model for that table. More specifically you're going to define a class whose instances will represent summits.

Edit the `workshopapp/model/__init__.py` file and change it so it looks like this:

```python
"""The appplication's model objects"""
from workshopapp.model.meta import Session, Base


def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    Session.configure(bind=engine)

    global Summit
    class Summit(Base):
        __tablename__ = 'summits'
        __table_args__ = {
            'autoload': True,
            'autoload_with': engine
            }
```

Setting `autoload` to `True` in the table arguments makes SQLAlchemy automatically discover the schema of the table (and load values for every column of the table when doing queries).

You can now restart the Paste web server (if not already started), and you should see the SQL commands SQLAlchemy sends to PostgreSQL for discovering the table columns:

```
(vp) $ paster serve --reload development.ini
```

2.6 Create controller

Here you are going to create a web service so information about summits can be requested through HTTP. For that a controller relying on the `summits` table, the `Summit` class really, will be created.

Pylons provides a command for generating controllers. You can use it to generate your `summits` controller:

```
(vp) $ paster controller summits
```

This command creates two files: `workshopapp/controllers/summits.py`, which includes the controller itself, and `workshopapp/tests/functional/test_summits.py`, which includes functional tests for that controller. These file are really just skeletons.

To check that your controller is functional you can now open http://localhost:5000/summits/index in your browser, you should get an `Hello World` HTML page.

You're now going to modify the `summits` controller so it returns a JSON representation of the first ten summits in the table. SQLAlchemy is used for querying the database, and a Pylons-specific decorator function (`jsonify`) is used to serialize the database objects into JSON.

Here the full code of the `summits` controller:

```
import logging

from pylons import request, response, session, tmpl_context as c, url
from pylons.controllers.util import abort, redirect
from pylons.decorators import jsonify
from workshopapp.model.meta import Session
from workshopapp.model import Summit

from workshopapp.lib.base import BaseController, render

log = logging.getLogger(__name__)

class SummitsController(BaseController):

    @jsonify
    def index(self):
        summits = []
        for summit in Session.query(Summit).limit(10):
            summits.append({
                "name": summit.name,
                "elevation": summit.elevation
                })


        return summits
```

2.7 Create new tables

This section shows how to create database tables when the Pylons application is set up.

To set up the project the `paster setup-app` command is used:

```
(vp) $ paster setup-app development.ini
```

This command executes the `setup_app` function defined in the `workshopapp/websetup.py` file.

By default the `setup_app` function calls the `Base.metadata.create_all` function. This function creates the tables defined in the model if they don't already exist in the database.

So to create tables at setup time you just need to declare new tables in the model.

Let's declare an `areas` table in the model. For that edit the `workshopapp/model/__init__.py` and change its content to:

```python
"""The appplication's model objects"""
from sqlalchemy.schema import Column
from sqlalchemy.types import Integer, String
from workshopapp.model.meta import Session, Base

def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    Session.configure(bind=engine)

    global Summit
    class Summit(Base):
        __tablename__ = 'summits'
        __table_args__ = {
            'autoload': True,
            'autoload_with': engine
            }

class Area(Base):
    __tablename__ = 'areas'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

The above code declares a table named `areas`. This table has two columns, one integer column named `id`, which is the primary key, and one string column named `name`.

You can now execute the `paster setup-app` command again. Once executed you should have the `areas` table in the PostgreSQL database. You can use `pgAdmin` or any other PostgreSQL client to verify the existence of that table.

Now you're going to modify the `setup_app` function to insert data into the `areas` table at setup time (recall that the `setup_app` function is defined in the `workshopapp/websetup.py` file):

```python
"""Setup the WorkshopApp application"""
import logging

import pylons.test

from workshopapp.config.environment import load_environment
from workshopapp.model.meta import Session, Base
from workshopapp.model import Area

log = logging.getLogger(__name__)
```

```
def setup_app(command, conf, vars):
    """Place any commands to setup workshopapp here"""
    # Don't reload the app if it was loaded under the testing environment
    if not pylons.test.pylonsapp:
        load_environment(conf.global_conf, conf.local_conf)

    # Create the tables if they don't already exist
    Base.metadata.create_all(bind=Session.bind)

    log.info("Adding default area...")
    default_area = Area()
    default_area.name = u"Default area"
    Session.add(default_area)
    Session.commit()
    log.info("Successfully set up.")
```

With the above code an area is added to the `areas` table. Execute the `paster setup-app` command again and verify that area was effectively inserted (for example using `pgAdmin` again).

Note: Here each time `paster setup-app` is executed a new area with the same name is inserted. Each inserted area has a different id though, the id is autoincremented, thanks to the sequence SQLAlchemy created in the database.

Bonus task 1

Create an `areas` controller, similar to the `summits` controller, but based on the `areas` table.

Bonus task 2

Add a new action to the `areas` controller (for example named `create`) for inserting new areas. You will use `Session.add` and `Session.commit` as in `websetup.py` for that.

CHAPTER

THREE

GEOALCHEMY

GeoAlchemy provides extensions to SQLAlchemy to work with spatial databases. GeoAlchemy currently supports PostGIS, MySQL, Spatialite, Oracle Locator, and Microsoft SQL Server 2008.

In this module you will learn how to use GeoAlchemy to interact with PostGIS.

More specifically, you will change the `Summit` model class to include the geometry column. You will also change the `summits` controller to return WKT or GeoJSON, to calculate buffers, and to create features in the database.

You'll also use the `geojson` and `Shapely` librairies.

3.1 Install

To install GeoAlchemy in the virtual Python environment, use:

```
(vp) $ easy_install "GeoAlchemy==0.4.1"
```

3.2 Change the model

To add support for the geometry column in your model, you need to declare it in the model class. GeoAlchemy provides a specific column class named `GeometryColumn` for the declaration of geometry colums.

Here is the updated code:

```
"""The application's model objects"""
from geoalchemy import GeometryColumn, Point
from workshopapp.model.meta import Session, Base


def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    Session.configure(bind=engine)

    global Summit
    class Summit(Base):
        __tablename__ = 'summits'
        __table_args__ = {
                'autoload': True,
                'autoload_with': engine
                }
        geom = GeometryColumn(Point)
```

When declaring the geometry column the geometry type must be set (`Point` here). This is particularly useful when relying on GeoAlchemy and SQLAlchemy for creating geographic tables.

3.3 Change the controller

It is now possible to export the geometries as WKT strings in the JSON.

```
import logging

from pylons import request, response, session, tmpl_context as c, url
from pylons.controllers.util import abort, redirect
from pylons.decorators import jsonify
from workshopapp.model.meta import Session
from workshopapp.model import Summit

from workshopapp.lib.base import BaseController, render

log = logging.getLogger(__name__)

class SummitsController(BaseController):

    @jsonify
    def index(self):
        summits = []
        for summit in Session.query(Summit).limit(10):
            summits.append({
                "name": summit.name,
                "elevation": summit.elevation,
                "wkt": Session.scalar(summit.geom.wkt)
                })


        return summits
```

Open http://localhost:5000/summits/index in your browser to see the JSON string that the `summits` controller now returns.

One thing to note is that each execution of `Session.scalar(summits.geom.wkt)` generates an SQL query. This is easily observable looking at the output of the `paster server` command.

To avoid these additional SQL queries you're going to use the Shapely library. So Shapely instead of PostGIS will be used to get a WKT representation of the geometry.

First install Shapely (version 1.2) with:

```
(vp) $ easy_install "Shapely==1.2"
```

You can now update the `workshopapp/controllers/summits.py` file with the following content:

```
import logging
import binascii
from shapely.wkb import loads

from pylons import request, response, session, tmpl_context as c, url
from pylons.controllers.util import abort, redirect
from pylons.decorators import jsonify
from workshopapp.model.meta import Session
from workshopapp.model import Summit
from workshopapp.lib.base import BaseController, render

log = logging.getLogger(__name__)

class SummitsController(BaseController):

    @jsonify
    def index(self):
        summits = []
        for summit in Session.query(Summit).limit(10):
            wkb = binascii.hexlify(summit.geom.geom_wkb).decode('hex')
            summits.append({
                "name": summit.name,
                "elevation": summit.elevation,
                "wkt": loads(str(summit.geom.geom_wkb)).wkt
                })

        return summits
```

Again you can open http://localhost:5000/summits/index in the browser and check the JSON string. It should be the same as previously.

Note: Doing performance tests here should show better performance when Shapely is used.

3.4 Output GeoJSON

In this section you're going to modify the `summits` controller again so the geographic objects returned by the controller are represented using the GeoJSON format. The `geojson` Python library will be used.

Start by installing the geojson library (version 1.0.1):

```
(vp) $ easy_install "geojson==1.0.1"
```

Now modify update the `workshopapp/controllers/summits.py` file with this content:

```
import logging
from shapely.wkb import loads
from geojson import Feature, FeatureCollection, dumps

from pylons import request, response, session, tmpl_context as c, url
from pylons.controllers.util import abort, redirect
from workshopapp.model.meta import Session
from workshopapp.model import Summit

from workshopapp.lib.base import BaseController, render

log = logging.getLogger(__name__)

class SummitsController(BaseController):

    def index(self):
        summits = []
        for summit in Session.query(Summit).limit(10):
            geometry = loads(str(summit.geom.geom_wkb))
            feature = Feature(
                    id=summit.id,
                    geometry=geometry,
                    properties={
                        "name": summit.name,
                        "elevation": summit.elevation
                        })
                summits.append(feature)

            response.content_type = 'application/json'
            return dumps(FeatureCollection(summits))
```

In the above code features are created from the objects read from the database. The `Feature` constructor is passed a Shapely geometry, which shows the integration between the Shapely and geojson libraries.

It is also interested to note that the `jsonify` decorator is no longer used, the `dumps` function from the geojson library is used instead.

3.5 Calculate buffer

Here you are going to extend the `summits` web service so it can return the buffer of a given feature. The URL will be `/summits/buffer/<id>` where `id` is the identifier of the feature for which the buffer is to be calculated.

To implement that you will add a `buffer` action to the `summits` controller. This action will retrieve the feature corresponding to the provided feature identifier, have PostGIS calculate the buffer of the feature, encode the resulting geometry in GeoJSON, and return the GeoJSON string.

You can now update the `workshopapp/controllers/summits.py` file with the following content:

```
import logging
from shapely.wkb import loads as wkbloads
from shapely.geometry import asShape
from geojson import GeoJSON, Feature, FeatureCollection, dumps, loads as geojsonloads
from geoalchemy import WKBSpatialElement, functions

from pylons import request, response, session, tmpl_context as c, url
from pylons.controllers.util import abort, redirect
from workshopapp.model.meta import Session
from workshopapp.model import Summit

from workshopapp.lib.base import BaseController, render

log = logging.getLogger(__name__)

class SummitsController(BaseController):

    def index(self):
        summits = []
        for summit in Session.query(Summit).limit(10):
            geometry = wkbloads(str(summit.geom.geom_wkb))
            feature = Feature(
                    id=summit.id,
                    geometry=geometry,
                    properties={
                        "name": summit.name,
                        "elevation": summit.elevation
                        })
            summits.append(feature)

        response.content_type = 'application/json'
        return dumps(FeatureCollection(summits))

    def buffer(self, id):
        buffer_geom = Session.query(
            functions.wkb(Summit.geom.buffer(10))).filter(Summit.id==id).first()
        if buffer_geom is None:
            abort(404)
        geometry = wkbloads(str(buffer_geom[0]))

        response.content_type = 'application/json'
        return dumps(geometry)
```

You can note that only one SELECT is done to the database with the above code. The same method could be applied to the `index` action.

Bonus task

Add an action named `buffer_shapely` that relies on Shapely instead of GeoAlchemy and PostGIS for the buffer calculation. And you can compare the performance obtained when using Shapely and when using PostGIS.

3.6 Create features

In this section you are going to create a web service allowing to add summits to the database.

For that you'll add a `create` action to the `summits` controller, which will parse the POST request, loads the GeoJSON feature, convert it to WKB with Shapely, and then create the feature in database using GeoAlchemy.

You can now update the `workshopapp/controllers/summits.py` file with the following content:

```
import logging
from shapely.wkb import loads as wkbloads
from shapely.geometry import asShape
from geojson import GeoJSON, Feature, FeatureCollection, dumps, loads as geojsonloads
from geoalchemy import WKBSpatialElement

from pylons import request, response, session, tmpl_context as c, url
from pylons.controllers.util import abort, redirect
from workshopapp.model.meta import Session
from workshopapp.model import Summit

from workshopapp.lib.base import BaseController, render

log = logging.getLogger(__name__)

class SummitsController(BaseController):

    def index(self):
        summits = []
        for summit in Session.query(Summit).limit(10):
            geometry = wkbloads(str(summit.geom.geom_wkb))
            feature = Feature(
                    id=summit.id,
                    geometry=geometry,
                    properties={
                        "name": summit.name,
                        "elevation": summit.elevation
                        })
            summits.append(feature)

        response.content_type = 'application/json'
        return dumps(FeatureCollection(summits))

    def buffer(self, id):
        buffer_geom = Session.query(
            functions.wkb(Summit.geom.buffer(10))).filter(Summit.id==id).first()
        if buffer_geom is None:
            abort(404)
        geometry = wkbloads(str(buffer_geom[0]))

        response.content_type = 'application/json'
        return dumps(geometry)

    def create(self):
        # read raw POST data
        content = request.environ['wsgi.input'].read(int(request.environ['CONTENT_LENGTH']))
        factory = lambda ob: GeoJSON.to_instance(ob)
        feature = geojsonloads(content, object_hook=factory)
        if not isinstance(feature, Feature):
            abort(400)
        shape = asShape(feature.geometry)
        summit = Summit()
        summit.geometry = WKBSpatialElement(buffer(shape.wkb))
        summit.elevation = feature.properties['elevation']
        summit.name = feature.properties['name']
        Session.add(summit)
        Session.commit()

        response.status = 201
        response.content_type = "application/json"
        feature.id = summit.id
        return dumps(feature)
```

Then you can make a test query to this new service using `curl` on the commandline:

```
$ curl http://localhost:5000/summits/create -d \
'{"geometry": {"type": "Point", "coordinates": [5.8759399999999999, 45.333889999999997]},
  "type": "Feature", "properties": {"elevation": 1876, "name": "Pas de Montbrun"}, "id": 2828}' \
-H 'Content-Type:"application/json"'
```

This command should output:

```
{"geometry": {"type": "Point", "coordinates": [5.8759399999999999, 45.333889999999997]},
"type": "Feature", "properties": {"elevation": 1876, "name": "Pas de Montbrun"}, "id": 5133}
```

You can check that a new summit has been created in database, using `pgAdmin` for example.

3.7 Create geographic tables

In the previous module you learned how to create tables with SQLAlchemy when setting up the application. GeoAlchemy makes it possible to create geographic tables.

You're going to convert the `areas` table into a geographic table, with a geometry column of type polygon, and have GeoAlchemy create that table in the database.

First, drop the `areas` table from the database, using pgAdmin for example.

Now update the `workshopapp/model/__init__.py` with this content:

```
"""The application's model objects"""
from sqlalchemy.schema import Column
from sqlalchemy.types import Integer, String
from geoalchemy import GeometryColumn, GeometryDDL, Point, Polygon
from workshopapp.model.meta import Session, Base


def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    Session.configure(bind=engine)

    global Summit
    class Summit(Base):
        __tablename__ = 'summits'
        __table_args__ = {
                'autoload': True,
                'autoload_with': engine
                }
        geom = GeometryColumn(Point)

class Area(Base):
    __tablename__ = 'areas'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    geom = GeometryColumn(Polygon)

GeometryDDL(Area.__table__)
```

This update implies:

- importing `GeometryDDL` and `Polyon` from `geoalchemy` (in addition to `GeometryColumn` and `Point`)
- declaring a geometry column of type `Polygon` in the `Area` class
- using `GeometryDDL(Area.__table__)` to make GeoAlchemy participate in the creation and destruction of the `areas` table

You can now execute the `paster setup-app` command and verify that the `areas` table and its geometry column are created:

```
(vp) $ paster setup-app development.ini
```

CHAPTER

FOUR

TILECACHE

This module shows how to use TileCache in a Pylons application, i.e. how to access TileCache through a controller of the application.

Doing this can be for example useful for securing access to TileCache, using the repoze.who/what security framework, or any other security framework that can be used in Pylons applications.

4.1 Install

To install TileCache in the virtual Python environment, use:

```
(vp) $ easy_install "TileCache==2.10"
```

4.2 Create configuration

First you'll need to define the TileCache configuration, i.e. where the cache is located on the file system, the layers to cache, etc. For that create a file name `tilecache.cfg` with the following content at the root of the application:

```
[cache]
type=Disk
base=/tmp/tilecache

[basic]
type=WMS
url=http://labs.metacarta.com/wms/vmap0
extension=png
```

In the `[cache]` section are defined the type and the filesystem location of the cache. With the `[basic]` section a layer name named `basic` is defined, this layer relies on a WMS. See http://tilecache.org/readme.html for more information about the configuration of TileCache.

4.3 Create controller

To create the controller for TileCache create the file `workshopapp/controllers/tilecache.py` with the following content:

```
from TileCache.Service import wsgiApp as TilecacheController
```

Note: Integrating TileCache in the application is as easy as this thanks to the WSGI interface TileCache implements (`wsgiApp`).

To be able to use this controller a specific *route* must be created. This route will define the matching between a URL and the `tilecache` controller. Edit the `workshopapp/config/routing.py` file and add the following line right after the `CUSTOM ROUTES HERE` comment:

```
map.connect('/tilecache', controller='tilecache')
```

You can now test your `tilecache` web service by opening the following URL in your browser:

```
http://localhost:5000/tilecache?LAYERS=basic&FORMAT=image%2Fpng&SERVICE=WMS&VERSION=1.1.1
&REQUEST=GetMap&STYLES=&EXCEPTIONS=application%2Fvnd.ogc.se_inimage&SRS=EPSG%3A4326
&BBOX=-11.25,33.75,0,45&WIDTH=256&HEIGHT=256
```

You should obtain a 256x256 image representing Spain.

4.4 View the map

For more fun you can also create a small OpenLayers (http://www.openlayers.org) application relying on your Tile-Cache web service.

Create `workshopapp/public/tilecache.html` with this content:

```
<head>
<style type="text/css">
    body {
        padding:0px;
        margin:0px
    }
    #map {
        width: 100%;
        height: 100%;
    }
</style>
<script src="http://openlayers.org/api/OpenLayers.js"></script>
<script type="text/javascript">
    var map, layer;

    function init(){
        map = new OpenLayers.Map('map', {'maxResolution': 360/512});
        layer = new OpenLayers.Layer.WMS(
            "Basic layer",
            "tilecache?", // the URL to the TileCache web service
            {layers: 'basic', format: 'image/png' }
        );
        map.addLayer(layer);
        map.zoomToMaxExtent();
    }
</script>
</head>
<body onload="init()">
  <div id="map"></div>
</body>
</html>
```

and open http://localhost:5000/tilecache.html in your browser.

Bonus task

Modify the `tilecache` controller in such a way that TileCache is executed in an action of the controller. You can look at http://pylonsbook.com/en/1.1/the-web-server-gateway-interface-wsgi.html#wsgi-in-pylons-controllers.

CHAPTER

FIVE

MAPFISH

MapFish is a flexible and complete framework for building rich web-mapping applications. It emphasizes high productivity, and high-quality development.

MapFish combines the power of Pylons, SQLAlchemy, GeoAlchemy, Shapely, and geojson.

In this module you will learn about what the MapFish framework provides, and how it simplifies the development of web services for querying and editing geographic features.

5.1 Install

To install MapFish in the virtual Python environment use:

```
(vp) $ easy_install "MapFish==2.0"
```

You should now have MapFish installed. You can check that using this command:

```
(vp) $ paster create --list-templates
```

This command should output this:

```
Available templates:
basic_package:   A basic setuptools-enabled package
mapfish:         MapFish application template
paste_deploy:    A web application deployed through paste.deploy
pylons:          Pylons application template
pylons_minimal:  Pylons minimal application template
```

Note the `mapfish` line.

5.2 Upgrade application

In this section you're going to convert your application from a Pylons application into a MapFish application. A MapFish application is a Pylons application with extra, MapFish-specific, features.

Edit `setup.py` and add `MapFish` to the list of paster plugins:

```
paster_plugins=['MapFish', 'PasteScript', 'Pylons'],
```

Now execute:

```
(vp) $ python setup.py egg_info
```

This registers MapFish as a paster plugin, which will allow you to use MapFish-specific commands for the development of the application. You will learn more about that later in this module.

You can check that your application is now a MapFish application by typing:

```
(vp) $ paster
```

And verify that the output of the command includes the following block:

```
mapfish:
  mf-controller   Create a MapFish controller and accompanying functional test
  mf-layer        Create a MapFish layer (controller + model).
  mf-model        Create a MapFish model
```

`mf-controller`, `mf-layer`, and `mf-model` are commands provided by the MapFish framework.

5.3 Create a MapFish web service

In this section you're going to create a MapFish web service that relies on the `summits` table. You will thereby understand what MapFish actually brings to the developer.

Create the file `layers.ini` at the root of your application, with this content:

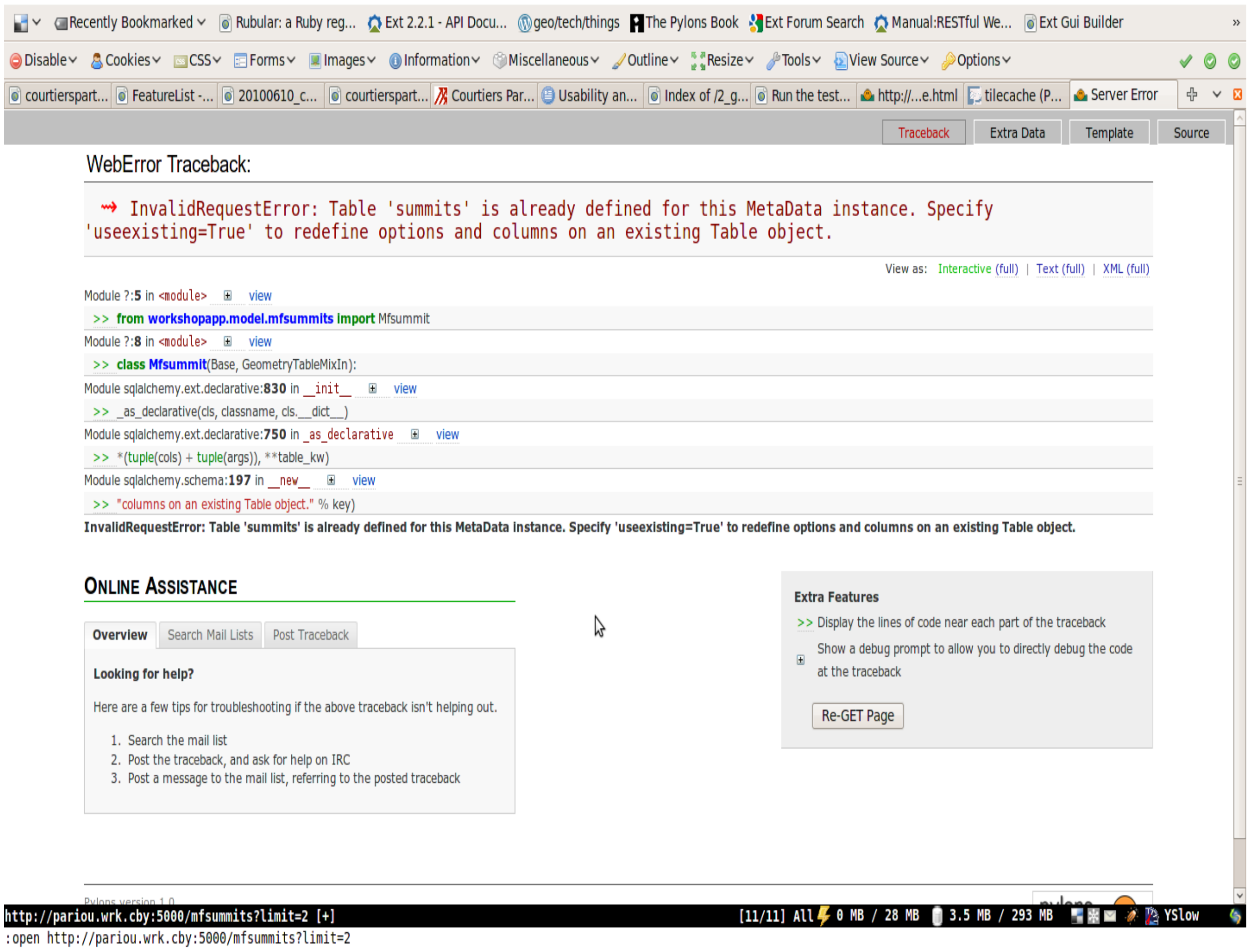
```
[mfsummits]
singular=mfsummit
plural=mfsummits
table=summits
epsg=4326
geomcolumn=geom
geomtype=Point
```

and execute this command:

```
(vp) $ paster mf-layer mfsummits


Creating /home/python/python_workshop/WorkshopApp/workshopapp/controllers/mfsummits.py
Creating /home/python/python_workshop/WorkshopApp/workshopapp/tests/functional/test_mfsummits.py

To create the appropriate RESTful mapping, add a map statement to your
config/routing.py file in the CUSTOM ROUTES section like this:

map.resource("mfsummit", "mfsummits")

Creating /home/python/python_workshop/WorkshopApp/workshopapp/model/mfsummits.py
```

As indicated in the command ouput edit the `workshopapp/config/routing.py` file and add an appropriate route by inserting `map.resource("mfsummit", "mfsummits")`. Insert this route before the default routes, for example right after the `tilecache` route.

You can now verify that your MapFish web service functions properly. For example open http://localhost:5000/mfsummits?limit=2 in FireFox. But in fact, because a model class already exists for the table `summits`, SQLAlchemy will produce an error:



WebError Traceback:

```
InvalidRequestError: Table 'summits' is already defined for this MetaData instance. Specify
'useexisting=True' to redefine options and columns on an existing Table object.
```

View as: Interactive (full) | Text (full) | XML (full)

```
Module ?:5 in <module>   view
>> from workshopapp.model.mfsummits import Mfsummit
Module ?:8 in <module>   view
>> class Mfsummit(Base, GeometryTableMixIn):
Module sqlalchemy.ext.declarative:830 in __init__   view
>> _as_declarative(cls, classname, cls.__dict__)
Module sqlalchemy.ext.declarative:750 in _as_declarative   view
>> *(tuple(cols) + tuple(args)), **table_kw)
Module sqlalchemy.schema:197 in __new__   view
>> "columns on an existing Table object." % key)
InvalidRequestError: Table 'summits' is already defined for this MetaData instance. Specify 'useexisting=True' to redefine options and columns on an existing Table object.
```

ONLINE ASSISTANCE

Overview | Search Mail Lists | Post Traceback

Looking for help?

Here are a few tips for troubleshooting if the above traceback isn't helping out.

1. Search the mail list
2. Post the traceback, and ask for help on IRC
3. Post a message to the mail list, referring to the posted traceback

Extra Features

>> Display the lines of code near each part of the traceback

Show a debug prompt to allow you to directly debug the code at the traceback

Re-GET Page

Pylons version 1.0

```
http://pariou.wrk.cby:5000/mfsummits?limit=2 [+]          [11/11] All 0 MB / 28 MB  3.5 MB / 293 MB  YSlow
:open http://pariou.wrk.cby:5000/mfsummits?limit=2
```

To correct the error edit `workshopapp/model/mfsummits.py` and add `useexisting: True` in the table arguments (`__table_args__`):

```
from sqlalchemy import Column, types

from geoalchemy import GeometryColumn, Point

from mapfish.sqlalchemygeom import GeometryTableMixIn
from workshopapp.model.meta import Session, Base

class Mfsummit(Base, GeometryTableMixIn):
    __tablename__ = 'summits'
    __table_args__ = {
        "autoload": True,
        "autoload_with": Session.bind,
        "useexisting": True
    }
    geom = GeometryColumn(Point(srid=4326))
```

Open http://localhost:5000/mfsummits?limit=2 again in the browser.

5.4 Play with the MapFish web service

MapFish implements HTTP interfaces for querying and editing features. Any MapFish web service, i.e. any web service created with the `paster mf-layer` command implements these interfaces. These interfaces are documented in http://trac.mapfish.org/trac/mapfish/wiki/MapFishProtocol.

5.4.1 Read

Here are examples of queries that you can try in the browser:

- http://localhost:5000/mfsummits/1
- http://localhost:5000/mfsummits/1?no_geom=true
- http://localhost:5000/mfsummits/1?no_geom=true&attrs=elevation
- http://localhost:5000/mfsummits?limit=10&offset=2
- http://localhost:5000/mfsummits?limit=10&order_by=elevation&dir=DESC
- http://localhost:5000/mfsummits?lon=86.86&lat=27.86&tolerance=5
- http://localhost:5000/mfsummits?bbox=85,26,87,28
- http://localhost:5000/mfsummits?limit=3&queryable=name,elevation&name__ilike=%col%&elevation__gte=1800

The above queries use the HTTP `GET` method.

5.4.2 Create

To create new features the HTTP `POST` method is used. As an example you can enter the following `curl` command in your terminal:

```
curl http://localhost:5000/mfsummits -X POST -H 'Content-Type:"application/json"' \
-d '{"type": "FeatureCollection",
     "features": [{"type": "Feature", "geometry": {"type": "Point", "coordinates": [5.8, 45.3]}}]}'
```

The command should output something like this:

```
{"type": "FeatureCollection",
 "features": [{"geometry": {"type": "Point",
                            "coordinates": [5.7999999999999998, 45.299999999999997]},
               "id": 5081, "type": "Feature",
               "bbox": [5.7999999999999998, 45.299999999999997,
                        5.7999999999999998, 45.299999999999997],
               "properties": {"elevation": null, "name": null}}]}
```

Note that GeoJSON response include the identifiers of the created features. In this example, one feature was created, its id is 5081.

Now open `http://localhost:5000/mfsummits/<id>` in the browser to get a GeoJSON representation of the freshly created feature, and verify that it has indeed been inserted in the database. `<id>` is to be replaced with the actual feature identifier (5081 in this example).

5.4.3 Update

To update a feature the HTTP `PUT` method is used. As an example use this `curl` command:

```
curl http://localhost:5000/mfsummits/<id> -X PUT -H 'Content-Type:"application/json"' \
-d '{"type": "Feature", "geometry": {"type": "Point", "coordinates": [6.0, 46]},
     "properties": {"name": "foo", "elevation": 1000}}'
```

Again replace `<id>` by the actual feature identifier (5081 in the above example).

The command should output something like this:

```
{"geometry": {"type": "Point", "coordinates": [6.0, 46.0]}, "id": 5081, "type": "Feature",
 "bbox": [6.0, 46.0, 6.0, 46.0], "properties": {"elevation": 1000, "name": "foo"}}
```

Again you can open `http://localhost:5000/mfsummits/<id>` in the browser to verify that the feature has been updated as expected.

5.4.4 Delete

To delete a feature the HTTP `DELETE` method is used. For example to delete with the `curl` command the feature whose id is 5081:

```
curl http://localhost:5000/mfsummits/<id> -X DELETE
```

If you open `http://localhost:5000/mfsummits/<id>` in the browser you should now get a 404 error.