Modular WebGIS Framework with Django and AngularJS

1) Python

Virtual Environment

Python packages can be separated into different virtual environments (provided by the *virtualenv* package).

http://docs.python-guide.org/en/latest/dev/virtualenvs/

Installation

```
pip install virtualenv
```

Create a new virtual environment (Linux)

```
cd project_folder
virtualenv venv
source venv/bin/activate
# pip and python executables are now used only for this virtual environment
```

Deactivate virtual environment after activation

```
deactivate
```

Store installed packages into a requirements-file

```
pip freeze > requirements.txt
```

Install packages from requirements.txt

```
pip install -r requirements.txt
```

2) Django

Introduction

Manual: https://docs.djangoproject.com/en/1.7/

Installation

```
pip install Django
```

Now test Django package in python:

```
import django
print(django.get_version())
```

First of all we need to create a project with django-admin.py:

```
django-admin.py startproject webgis
```

```
# new folder and files created
# - webgis
#   - webgis
#     - settings.py
#     - urls.py
#     - wsgi.py
#   - db.sqlite3
#   - manage.py
```

Afterwards we can rename the root-folder of the created structure "webgis" to "project" (for example). Finally there are two directories in the workspace/project folder:

```
# - venv          virtual environment
# - project       django project
```

To initialize the database we have to synchronize the database first:

```
cd project
python manage.py syncdb
```

Finally we have to create an admin superuser:

```
python manage.py createsuperuser
```

Settings (settings.py)

Relevant parts of *webgis/settings.py* for the development are following variables:

- INSTALLED_APPS
- STATIC_URL
- DATABASES
- SUIT_CONFIG

Usage

With manage.py several tasks can be executed for django:

```
# run server
python manage.py runserver
# create new app
python manage.py startapp <appname>
# update changed models
python manage.py makemigrations
python manage.py migrate
```

Deployment

/var/www/webgis.essi-services.net/index.wsgi

```
import os
import sys
import site
# Add site-packages of the chosen virtualenv to work with
site.addsitedir('/home/jonas/workspace/webgis/venv/local/lib
/python2.7/site-packages')
```

```
# Add the app's directory to the PYTHONPATH
sys.path.append('/home/jonas/workspace/webgis/project')
sys.path.append('/home/jonas/workspace/webgis/project/webgis
')
os.environ['DJANGO_SETTINGS_MODULE'] = 'webgis.settings'

# Activate the virtual environment
activate_env=os.path.expanduser("/home/jonas/workspace/webgi
s/venv/bin/activate_this.py")
execfile(activate_env, dict(__file__=activate_env))

# start django handler
import django.core.wsgi
application = django.core.wsgi.get_wsgi_application()
```

Apache Configuration file

```
<VirtualHost 37.59.98.101:80>

   ServerAdmin jonas.eberle@eberle-mail.de
   ServerName  webgis.vps135108.ovh.net
   ServerAlias webgis.vps135108.ovh.net

   WSGIScriptAlias / /var/www/webgis.essi-services.net/index.wsgi

   Alias /static/ /var/www/webgis.essi-services.net/static/

   <Location "/static/">
       Options –Indexes
   </Location>

</VirtualHost>
```

Plugins

Installed plugins (requirements.txt):

```
Django==1.7.5
django-allauth==0.19.1
django-cronjobs==0.2.3
django-filter==0.9.2
django-rest-auth==0.3.4
django-suit==0.2.12
djangorestframework==3.0.5
geojson==1.0.9
oauthlib==0.7.2
OWSLib==0.8.13
python-dateutil==2.4.0
python-openid==2.2.5
pytz==2014.10
requests==2.5.3
requests-oauthlib==0.4.2
six==1.9.0
```

Authorization:
- django-allauth: http://django-allauth.readthedocs.org/en/latest/
- django-rest-auth: http://django-rest-auth.readthedocs.org/en/latest/

Admin interface:
- django-suit: http://djangosuit.com/

Rest framework:
- djangorestframework: http://www.django-rest-framework.org/

Spatial data handling:
- geojson: https://github.com/frewsxcv/python-geojson
- OWSLib: http://geopython.github.io/OWSLib/

Database Migration
http://matthewwittering.com/blog/how-to-migrating-the-database-engine-for-django.html

1) Dump data

```
python manage.py dumpdata > database.json
```

2) Modify settings.py

```
#####################################################
# Database
https://docs.djangoproject.com/en/1.7/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

3) Create new database structures

```
python manage.py dumpdata > database.json
```

4) Empty the new database

```
python manage.py sqlflush
```

5) Import data

```
python manage.py loaddata database.json
```

Development

Apps
A Django project always consists of a “project” and several “apps”. The initial project contains the settings-file and at leas the urls-file to define the urls from project and integrated apps. Any installed app must be listed in the INSTALLED_APPS variable (see settings.py).

A new app can be created with the following command:

```
python manage.py startapp <appname>
```

Models

A model is a source and representative view of data. Models are used to store data for different content types. Several field types are available (Char, Integer, Float, Relations, File). When the model is registered to the Django backend, the content can be added through it. Several methods are available to filter the stored objects.

Example:

```
from django.db import models

class BaseLayer(models.Model):
    title = models.CharField(max_length=200)
    url = models.CharField(max_length=200, blank=True,
                           null=True)
    layername = models.CharField(max_length=200, blank=True,
                                 null=True)
    type = models.CharField(max_length=10, choices=[
           ('BingMaps', 'BingMaps'), ('WMS', 'WMS'),
           ('WMTS', 'WMTS'), ('MapQuest', 'MapQuest')])

    def __unicode__(self):
        return self.title
```

These models are converted to a database structure when executing the following commands:

```
python manage.py makemigrations

python manage.py migrate
```

Further information can be found here:
https://docs.djangoproject.com/en/1.7/topics/db/models/

Views

Any output from Django is related to a view. Models can be requested, filtered and published based on HTML, Templates or other formats.

Function-based view:

```
from django.http import HttpResponse
from django.template import RequestContext, loader
from mapviewer import models

def index(request):
    viewers = models.MapViewer.objects.all()
    template = loader.get_template('index.html')
    context = RequestContext(request, {
        'viewers': viewers,
    })
    return HttpResponse(template.render(context))
```

Class-based view:

```
from django.http import HttpResponse
from django.views.generic import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

Authentication within views:

```
# Does user is logged in?
request.user.is_authenticated()

# List all groups of the authenticated user
request.user.groups.all()

# Get username, id, email from user
request.user.id, request.user.username, request.user.email
```

URLs

Any view has to be linked from an URL-pattern that is registered in the urls.py. URL-patterns consist of a regular expression and a view or an include-link. An include-link can link to another urls.py from an app.

Project example (webgis/urls.py):

```
urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^authapi/', include('authapi.urls')),
    url(r'^mapviewer/', include('mapviewer.urls')),
    url(r'^layers/', include('layers.urls')),
    url(r'^csw/', include('csw.urls')),
    url(r'^admin/', include(admin.site.urls)),
) +
static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

App example (layers/urls.py):

```
from django.conf.urls import patterns, url, include
from . import views

urlpatterns = patterns('',
    url(r'^list/$', views.LayerList.as_view()),
    url(r'^detail/(?P<pk>[0-9]+)$',views.Detail.as_view()),
    url(r'^info$', views.LayerInfo.as_view()),
    url(r'^data$', views.DataRequest.as_view()),
    url(r'^sos/stations$', views.GetSOSStations.as_view()),
    url(r'^sos/data$', views.GetSOSObservation.as_view()),
)
```

Admin

First of all a Model has to be registered to be available in the Django backend:

```
from django.contrib import admin
from .models import CSW

admin.site.register(CSW)

# Models from installed plugins can also be unregistered
with admin.site.unregister(MODELNAME)
```

Furthermore the admin and the form (to save/create new entries) can be adjusted, for example for adding own error messages in the form:

```
from django import forms
from django.contrib import admin
from .models import BaseLayer

class BaseLayerForm(forms.ModelForm):
    def clean(self):
        cleaned_data = super(BaseLayerForm, self).clean()
        type = cleaned_data.get('type')
        url = cleaned_data.get('url')
        layername = cleaned_data.get('layername')

        if type == 'WMS' and (url == '' or layername == ''):
            self.add_error('url', 'URL has to be specified')
            self.add_error('layername', 'Layername has to be
                                        specified')

class BaseLayerAdmin(admin.ModelAdmin):
    form = BaseLayerForm

admin.site.register(BaseLayer, BaseLayerAdmin)
```

Rest framework

In the Debug mode the Rest framework provides an interface for testing the provided rest-based views. You can view the interface when linking to the URL that is related to the view.

Serializer:

A Serializer based on a model is needed to provide a JSON output within the rest framework. With a Serializer we can specify individual fields for the output. Furthermore we can create new fields (see legend variable in example) based on other field values.

```
from rest_framework import serializers
from .models import Layer

class LayerSerializer(serializers.ModelSerializer):
    legend = serializers.FileField(source='legend_graphic')
             or serializers.CharField(source='legend_url')

    class Meta:
        model = Layer
        fields = ('id', 'title', 'abstract', 'ogc_link',
```

```
                    'ogc_layer', 'ogc_type', 'west', 'east',
                    'north', 'south', 'epsg', 'downloadable',
                    'legend', 'download_link',
                    'download_layer', 'download_type')
```

APIView:
The APIView class from the Django rest framework provides a structure to publish an HTTP-REST-interface with GET, POST, PUT, DELETE methods. When a function of these methods is available, it can be execute by the related HTTP request. The Response class need a JSON object as parameter. You don't need to use a serializer, you can also generate your own JSON object.

```
from rest_framework.views import APIView
from rest_framework.response import Response
from .models import Layer, LayerSerializer

class LayerList(APIView):

    def get(self, request):
        layers = Layer.objects.all()
        serializer = LayerSerializer(layers, many=True)
        return Response(serializer.data)
```

3) AngularJS

Introduction

Homepage: https://angularjs.org/
Tutorial: https://docs.angularjs.org/tutorial
API-Reference: https://docs.angularjs.org/api

Development

For the development of the modular WebGIS framework the following parts are important to know.

Service

Services can be used to organize and share code and variables across the app. A service consists of an object (here: `var sharedService`) that is returned. Within the object we can define variables and functions. With the name of the function (here: AlertService) that service can be used in other tools (e.g., controller, directive).

With the $rootScope object we can define broadcasts to send events to other parts within the AngularJS app.

Example AlertService:

```
angular.module('webgisApp')
 .service('AlertService', function AlertService($rootScope){
    var sharedService = {
        alert: null,
        'addAlert': function (alert) {
            this.alert = alert;
```

```
            $rootScope.$broadcast('alert.added');
        }
    }
    return sharedService;
 })
```

Controller

An AngularJS controller binds specific JavaScript code to a certain HTML code. Within the HTML code the attribute "ng-controller" needs to be specified with a name of the controller. Within the AngularJS controller function ".controller" variables and functions can be defined to the $scope object. Variables can be used within the HTML block of the controller, by specifying the template code "{{variablename}}" or by using AngularJS specific options, e.g. the "ng-repeat" attribute. See the api-directive reference for further possibilities: https://docs.angularjs.org/api/ng/directive

Example AlertController:

```
<div id="alerts" ng-controller="AlertCtrl">
   <alert ng-repeat="alert in alerts"
          type="{{alert.type}}"
          close="closeAlert($index)"
   >
         {{alert.msg}}
   </alert>
</div>
```

```
angular.module('webgisApp')
 .controller('AlertCtrl', function ($scope, AlertService){
    $scope.alerts = [];

    $scope.$on('alert.added', function () {
        $scope.alerts = [AlertService.alert];
    });

    $scope.closeAlert = function (index) {
        $scope.alerts = [];
    };
 })
```

Example for input fields:

```
<div ng-controller="SearchBoxCtrl">
  <form ng-submit="search();">
    <input type="text" ng-model="text">
    <button ng-click="search();">Search</button>
  </form>
</div>
```

```
angular.module('webgisApp')
 .controller('SearchBoxCtrl', function($scope){
```

```
    $scope.text = '';

    $scope.search = function() {
        alert('You searched for '+$scope.text);
    };
 })
```

Directive

An AngularJS directive is used to execute code when specific HTML code occurs. A directive can be restricted to a specific attribute (A), to an element (E) or to classes (C). These restrictions can also be combined. A "link" function can be returned within the directive to execute JavaScript code for any element that is affected.

Example transparency slider for current layers:

```
<input transparency-slider class="slider" type="text" data-
slider-min="0" data-slider-max="100" data-slider-step="1"
data-slider-value="100"/>
```

```
angular.module('webgisApp')
 .directive('transparencySlider', function () {
   return {
      restrict: 'A',
      link: function(scope, element) {
          element.slider({})
      }
   };
 })
```

See following link for further information:
https://docs.angularjs.org/guide/directive

Routeprovider

The routing is used to link through controllers and views through an anchor in the URL. Therefore the plugin "ngRoute" has to be used and defined in the app.

Example for Alerting and Password reset:
In the first example ('/verify_success') we define a controller being executed, and two variables (alertType, alertMsg). These variables will be available in the controller within the $scope.current object. You can execute this example by following the links:
http://webgis.essi-services.org/mapviewer/detail/1.html#/verify_success

In the second example ('/password-reset-confirm') the variables (uid, token) are provided in the URL. So we can specify a URL with an anchor and variables that can be used later on in the controller within the $routeParams object.
Example URL: http://webgis.essi-services.org/mapviewer/detail/1.html#/password-reset-confirm/uid/token

```
angular.module('webgisApp', [
    'ngRoute'
```

```
])
 .config(function ($routeProvider) {
    $routeProvider
        .when('/verify_success', {
            template: '',
            controller: 'RouteAlertCtrl',
            alertType: 'success',
            alertMsg: 'Verification successful.'
        })
        .when('/password-reset-confirm/:uid/:token', {
            template: '',
            controller: 'NewPwCtrl'
        });
 })
 .controller('RouteAlertCtrl', function ($scope, $route,
              AlertService) {
    AlertService.addAlert({'type': $route.current.alertType,
                           'msg': $route.current.alertMsg});
 })
 .controller('NewPwCtrl', function ($scope, $routeParams) {
    $scope.uid = $routeParams.uid;
    $scope.token = $routeParams.token;
 })
```

Plugins

Plugins need to be specified in the first occurrence of the AngularJS app. We use plugins for cookies, resource (for restful http services), routing, ui-bootstrap, drag and drop lists, and popovers.

```
angular.module('webgisApp', [
   'ngCookies',
   'ngResource',
   'ngRoute',
   'ui.bootstrap',
   'dndLists',
   'nsPopover',
])
```

4) Project-specific developments

Django Suit Admin

Homepage: http://djangosuit.com/
Documentation: http://django-suit.readthedocs.org/en/develop/

Admin menu configuration

```
SUIT_CONFIG = {
 'MENU': (
     {'app': 'auth', 'label': 'Authorization', 'icon':'icon-
       user', 'models': ('user', 'group')},
```

```
        {'app': 'mapviewer', 'label': 'Mapviewer', 'models':
          ('mapviewer', 'template', 'baselayer')},
        {'app': 'layers', 'label': 'Layers', 'models':
          ('layergroup', 'layer', 'contact')},
        {'app': 'csw', 'label': 'Suche', 'models': ('csw')},
    )
}
```

Admin extensions (inline tabular, form tabs)

Django Suit also extends the admin backend with further support for inline tabular like we did this to define the layergroups. Furthermore the admin forms can be changed to a form within different tabs. See layers/admin.py and the Suit documentation page for further information.

Modals

Based on: http://angular-ui.github.io/bootstrap/

Simple modal popups can be defined with the $modal object and the pre-configured ModalInstanceCtrl controller. This controller provides a popup-close function and the possibility to pass data to the template.

When executing a modal popup you have to specify the popup template:

```
var modalInstance = $modal.open({
    controller: 'ModalInstanceCtrl',
    template: '<div modal-draggable class="modal-
header"><h1>{{title}}</h1></div><div class="modal-body
metadataModal" ng-
include="\'/static/includes/metadata.html\'"></div><div
class="modal-footer"><button class="btn btn-primary" ng-
click="close()">Close</button></div>',
    resolve: {
        data: function() {return data;},
        title: function() {return data.title;}
    }
});
```

Bootbox

Based on: http://bootboxjs.com/

Bootbox can be used to provide, e.g., Alert, Confirm and Dialog popups:

```
bootbox.alert("Hello world!", function() {
  Example.show("Hello world callback");
});

bootbox.confirm("Are you sure?", function(result) {
  Example.show("Confirm result: "+result);
});
```

```
bootbox.dialog({
  title: "That html",
  message: '<img src="logo.png"/><br/> You can use <b>html</b>'
  buttons: {
    success: {
      label: "Success!",
      className: "btn-success",
      callback: function() {
        Example.show("great success");
      }
    }
  }
});
```

HTTP Requests

The djangoRequests service is a pre-configured AngularJS service that provides a central request function to execute HTTP Rest calls. If success, the first function of "then" is being executed with the resulting data.

```
djangoRequests.request({
  'method': "GET",
  'url': '/layers/detail/'+layer.id+'.json'
}).then(function(data){
    console.log(data);
}, function(error) {
    bootbox.alert('<h1>Error</h1>');
})
```

Folder & file structure

- authapi (django app)
  - templates/account/email
    - text files for email messages
- csw (django app)
- layers (django app)
- mapviewer (django app)
- media (files uploaded through Django admin, legend graphics)
- static
  - css
  - includes
    - auth-register.html
    - auth.html
    - climatechart.html

        - info.html
        - metadata.html
        - searchresults.html
        - settings.html
    - js
        - app.js
        - auth
            - auth.js
        - csw
            - csw.js
        - map
            - map.js
    - lib
        - angularjs files
        - bootbox
        - bootstrap js
        - dygraph files
        - jquery
        - ol.js (OpenLayers 3)
    - further files from django plugins used in the project

- templates
    - index.html (overview auf mapviewer)
    - viewer.html (detailed webgis viewer)

- webgis (django project)

HTTP REST services

Authentication

- authapi
    - rest
        - login (Login user)
        - logout (Logout user)
        - user (Get information from currently logged in user)
        - user/delete (Delete currently logged in user)
        - registration/<mapviewerID> (User registration)
        - registration/confirm-email (Confirm user registration)
        - password
            - change (Change password for current user)
            - reset (Reset password via mail)
            - reset/confirm (Confirm password reset)

Mapviewer

- mapviewer
    - detail
        - <id>.json (Get mapviewer settings & data as JSON)
        - <id>.html (Get mapviewer as HTML)

Layers

- layers
  - detail
    - <id>.json (Get metadata for layer as JSON)
    - <id>/download (Redirect to download file)
  - info (Proxy for WMS getFeatureInfo requests)
  - data (Proxy for external services, e.g., WFS server)
  - capabilities
    - time (Convert time interval to dates)
    - WMS (Parser for WMS GetCapabilities request)
    - WMTS (Parser for WMTS GetCapabilities request)
  - sos
    - <layerid>
      - stations (List stations from SOS server as GeoJSON)
    - data (Get observation data from SOS as JSON)

CSW

- csw
  - search
    - <id> (Search within external CSW server)

Cronjobs

Cronjobs have been introduced to cache layer-specific settings. The have to be registered in the cronjobs settings of the server.

Django-cronjobs is used to easily integrate and provide cronjob capabilities with Django apps.

Example from layers/cron.py:

```
import cronjobs
from .models import Layer

#cronjob to cache_sos_layers
@cronjobs.register()
def cache_sos_layers():
    layers = Layer.objects.filter(ogc_type__exact='SOS')
    for layer in layers:
        print 'Layer: '+str(layer.id)
        layer.cache()
```

They can be executed with the following command:

```
python manage.py cron cache_sos_layers
```

No further cronjobs have been added so far.

Signals

Django signals can be used to trigger events. A post_save signal has been implemented to cache OGC SOS layers after the laver was saved.

Example from layers/models.py:

```
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=Layer)
def post_save_layer_sos(sender, instance, **kwargs):
    if instance.ogc_type == 'SOS':
        instance.cache()
```

Implementation structure