foss4g_2017_geonode_solr

Building SDIs and geoportals with GeoNode and a search engine

Organizers

- Paolo Corti - Center for Geographic Analysis, Harvard University
- Ben Lewis - Center for Geographic Analysis, Harvard University
- Devika Kakkar - Center for Geographic Analysis, Harvard University

Abstract

WorldMap is an SDI/GeoPortal project running since 2010 by the Center for Geographic Analysis, Harvard University, that exposes 25k+ internal layers, 200k+ remote layers, 7k+ maps and used by 20k+ registered users. It is built on top of GeoNode and its core components (Django, GeoServer, GeoWebCache, PostgreSQL/PostGIS, pycsw).

Users can build maps with a web browser using the internal layers (stored in the PostGIS database and the filesystem and exposed by GeoServer as OGC web services) and the remote layers, exposed by external OGC and Esri Rest Web Services. Remote layers are harvested, health checked, processed and exposed with a system based on Django and MapProxy, named Harvard Hypermap (HHypermap). A search engine, based on Solr or Elasticsearch, enriches the WorldMap user's experience providing powerful features such as keyword, spatial and temporal facets, full text search, natural language processing, weighted results and much more.

The organizers will lead the workshop's participants, with a series of step by step tutorials, to setup a development environment containing all the needed components, to get a basic understanding of the solution stack involved to replicate such an architecture in their own geoportals/SDIs implementation, and to discuss optimization techniques for taking advantage of the solution for handling a large number of datasets.

The workshop is targeted to geospatial developers who should have a basic understanding of web development, OGC standards and spatial databases.

Prerequisites

You need to take your notebook at the workshop. The notebook should have at least 4GB of RAM. You can use any operating system (Linux, OS X, Windows), provided that you install the following tools:

- git
- Vagrant
- VirtualBox
- QGIS
- Firefox

Note: you could use Chrome in place of Firefox for all of the tutorials of this workshop. Though, one tutorial will use the Firefox Developer Tools to inspect requests to the OGC services. If using Chrome you should be able to follow along that tutorial using the Chrome Developer Tools

Outline of tutorials

- Introduction
- Setup the workshop environment
- A GeoNode quickstart
- Programming GeoNode with Python
- OGC services with GeoServer
- Tiles caching with GeoWebCache
- Catalogue services with pycsw
- Spatial queries with PostGIS
- Using GeoNode OGC services with external clients
- Using GeoNode management commands
- Using a search engine with GeoNode: a quick tour of Solr
- Developing a custom application for GeoNode
- Running asynchronous tasks using a task queue (Celery/RabbitMQ)

foss4g_2017_geonode_solr

Setup the workshop environment

In this first step of the workshop you will create the workshop development environment, running an instance of GeoNode and one instance of Solr. For this purpose you will create a virtual machine with Ubuntu 16.04 LTS using Vagrant, and you will install all of the needed components to have a working environment for the aims of the workshop.

At the end of this step you will have an Ubuntu box with operational instances of GeoNode and Solr.

Git clone the workshop repository

Open your shell and clone the workshop repository:

```
$ git clone https://github.com/capooti/foss4g_2017_geonode_solr.git
```

Start the Vagrant box and SSH into it

Vagrant is a command line utility for managing virtual machines. You will use it to manage the virtual machine needed in this workshop.

For what is needed for this workshop, you just need to start the Vagrant box and login into it using SSH:

```
$ cd foss4g_2017_geonode_solr
$ vagrant up
$ vagrant ssh
```

If you want more information about Vagrant, a good introduction is here

SSH, which stands for Secure Shell, creates a channel for running a shell on a remote computer, with end-to-end encryption between the local and the remote computer.

If you are new to SSH, a good introduction can be found here

GeoNode installation

In this step of the workshop you will install GeoNode, on top of which you will build a testing geoportal in the next tutorials of the workshop.

GeoNode is an Open Source, Content Management System (CMS) for geospatial data. It is a web-based application and platform for developing Geographic Information Systems (GIS) and for deploying Spatial Data Infrastructures (SDI). For more information about it check its documentation, the next tutorials of this workshop or both.

GeoNode it is based on Django, which is a great Python web framework. Other core components of GeoNode are PostgreSQL/PostGIS, GeoServer, pycsw or GeoNetworks and GeoExplorer.

Install requirements

You will use the Ubuntu apt package manager to install the requirements for the environment.

You will install Python 2.7 (needed by GeoNode, which does not run on Python 3, the default Python version at Ubuntu 16.04) and all of the other requirements, including Java, GDAL and RabbitMQ:

```
$ sudo apt-get update
$ sudo apt install python virtualenv python-dev build-essential gdal-bin libgdal-dev
$ sudo ln -s /usr/lib/x86_64-linux-gnu/libproj.so.9 /usr/lib/x86_64-linux-gnu/libproj
```

GeoServer and Solr both run on top of a Java Runtime Environment. default-jre is the official Ubuntu JRE: make sure java is working by running the java -version command:

```
$ java -version
openjdk version "1.8.0_131"
OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-0ubuntu1.16.04.2-b11)
OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)
```

Create a virtual environment

GeoNode is based on Django, which is a Python web framework. It is generally a good idea to install all of the Python GeoNode requirements separately from the global system requirements. A great way to do this is using a tool like virtualenv.

If you are new to virtualenv you may take some minutes to read the documentation

For the aims of the workshop, create a Python virtual environment and activate it:

```
$ cd /workshop/
$ virtualenv --python=python2 env
$ . env/bin/activate
```

Install GeoNode

You will install GeoNode 2.6.1 from GitHub. Here is what you need to do:

```
$ git clone https://github.com/GeoNode/geonode.git
$ cd geonode
$ git checkout -b 2.6.1 tags/2.6.1
$ pip install -e .
$ pip install pygdal==1.11.3.3
$ pip install django-jsonfield-compat
$ pip install ipython
$ paver setup
$ paver sync
```

ipython is not required by GeoNode, but it will be a very helpful tool for the purpose of this workshop.

Start GeoNode and GeoServer

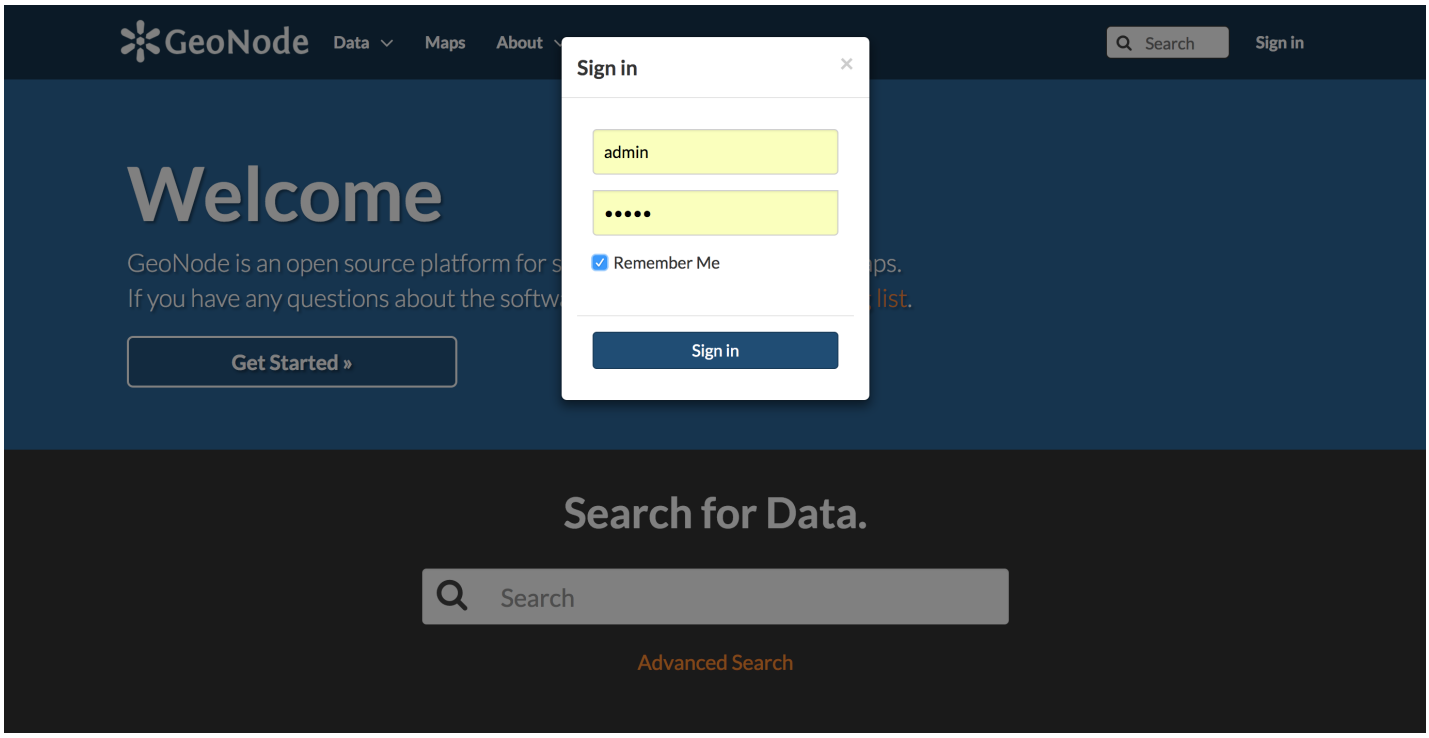
If everything went smoothly, you should now be able to start GeoServer and GeoNode:

```
$ paver start_geoserver
$ python manage.py runserver 0.0.0.0:8000
```

Now try to access to the GeoNode instance at: http://localhost:8000/

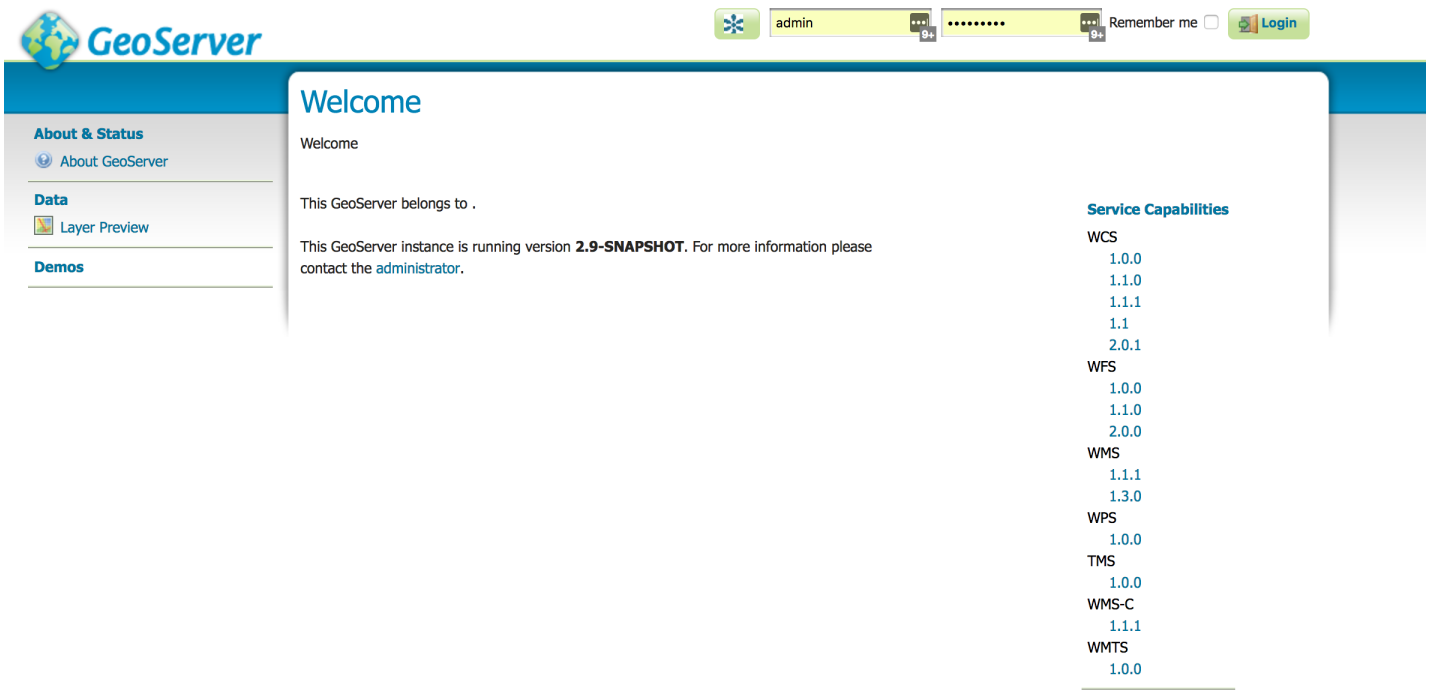
Log in in GeoNode using the administrative account which should have been already created:

- user: admin
- password: admin



Now click on the "admin" (or whatever is the username you created for the superuser) link, and then on GeoServer.

The GeoServer administrative interface should correctly display at the url: http://localhost:8080/geoserver/web/



Click on the snowflake-like button to the left of username to log in to GeoServer

Solr installation

Apache Solr is an open source enterprise search platform, developed in Java, on top of the Apache Lucene project. Its major features include full-text search, hit highlighting, faceted search, real-time indexing, dynamic clustering, database integration, NoSQL features and rich document (e.g., Word, PDF) handling. Providing distributed search and index replication, Solr is designed for scalability and fault tolerance.

Solr provides a great spatial support as well, which can be used for example for providing spacial facets to your SDI. Solr is, with Elasticsearch, the most popular enterprise search engine nowadays.

NOTE: the workshop uses Solr as the search engine, but it should be relatively easy to switch the search engine to Elasticsearch.

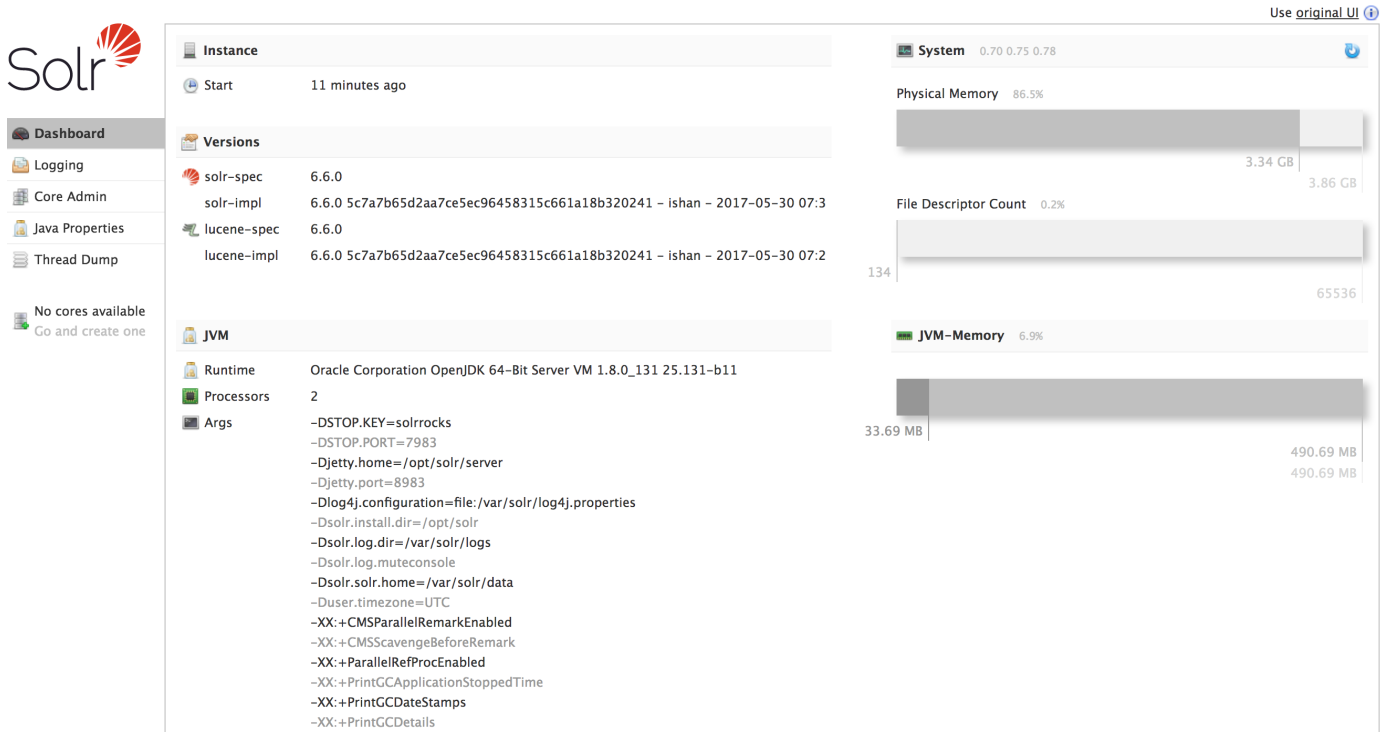
As a first thing, download and extract the latest Solr release in your Vagrant box. Then install the service following these instructions:

(Open a new terminal window if geonode is running).

```
$ cd /opt/
$ sudo wget http://apache.claz.org/lucene/solr/6.6.0/solr-6.6.0.tgz
$ sudo tar zxf solr-6.6.0.tgz
$ sudo tar xzf solr-6.6.0.tgz solr-6.6.0/bin/install_solr_service.sh --strip-componen
$ sudo ./install_solr_service.sh solr-6.6.0.tgz
```

Now the Solr service should be already up and running.

Make sure that the Solr administrative interface is correctly running at http://localhost:8983/solr/#/



PostgreSQL and PostGIS installation

By default, doing the GeoNode setup from git as you did, the Django database is created on sqlite, and the vector datasets (shapefiles) are uploaded as shapefile stores in GeoServer.

In production it is desirable to use a relational database in place of sqlite for the Django database. At the same time vector files are better stored in a spatial database: concurrent access will be optimized and transactions will be enabled - which means that GeoNode users will be able - if needed - to edit remotely the vector datasets using the GeoExplorer edit toolbar.

PostgreSQL, thanks to its wonderful spatial support provided by PostGIS, is by far the best choice for a RDBMS in GeoNode.

Install PostgreSQL and PostGIS from apt packages:

```
sudo apt-get install postgresql postgresql-contrib postgis
```

Summary of URLs

| Application | URL | | ——— | ——— | | GeoServer | http://localhost:8080/geoserver/web/ | | GeoNode | http://localhost:8000/ | | Solr | http://localhost:8983/solr/#/ |

Troubleshooting

If you get an “Address already in use” error, you may find this stackoverflow to be helpful.

foss4g_2017_geonode_solr

A GeoNode quickstart

In this first tutorial of the workshop you will use GeoNode as an end user. You will upload some layers using the user interface, then you will compile metadata for these layers and create a map with them.

You will use the datasets you are uploading to GeoNode in the next workshop's tutorials, so make it sure to follow these instructions carefully.

You can find the shapefiles that are needed for this tutorial in the *data/shapefiles* directory. They are zipped, and you don't need to unzip them as GeoNode can import zipped shapefiles.

Overview of the datasets

The datasets you are using here are part of the WorldMap Boston Research Map.

> Harvard WorldMap is an online, open source mapping platform developed to lower barriers for scholars who wish to explore, visualize, edit, and publish geospatial information.
> The WorldMap platform is based on an open source stack, including GeoNode, and it is developed and maintained by Harvard CGA

> WorldMap Boston Research Map is an open source web mapping system that is an ongoing project of the Boston Area Research Initiative (BARI) in conjunction with the WorldMap team and built on top of the WorldMap platform. It is intended to help faculty and their students, policymakers and practitioners, and community members to explore the neighborhoods of Boston from their computer. Visitors to BostonMap can:
>
> 1. Interact with the best available public data for the Boston region, while also uploading their own data.
> 2. See the whole Boston area but also zoom in to particular places.
> 3. Accumulate both contemporary and historical data supplied by researchers and make it permanently accessible online.
> 4. Work collaboratively across disciplines and organizations with spatial information in an online environment.

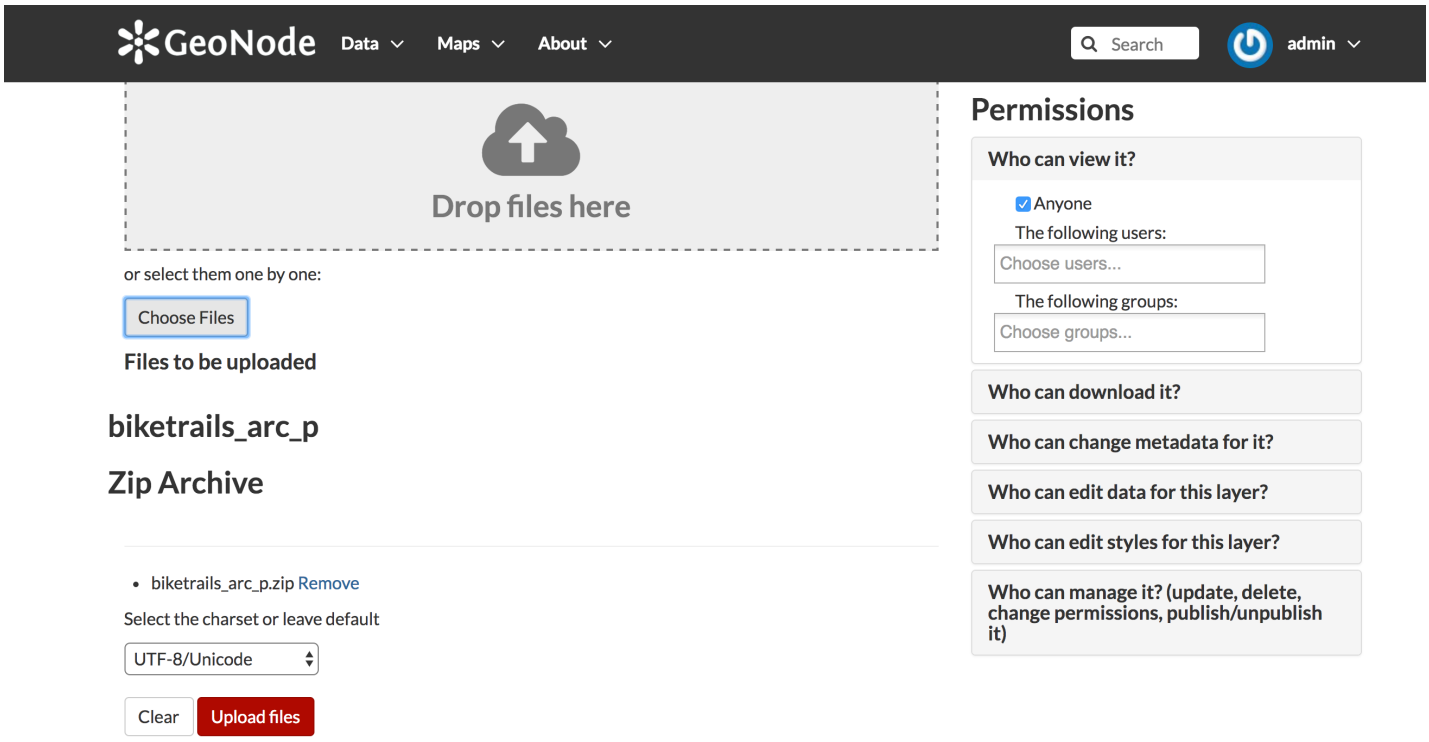
Here is a quick overview of the datasets you are going to use:

- *biketrails_arc_p*: bike trails in Boston (2009)
- *boston_public_schools_2012_z1l*: all schools in the Boston Public Schools District (2012)
- *socioeconomic_status_2000_2014_9p1*: Illustrates a measure of socio-economic status based on US Census indicators, as well as trends in socieconomic status change. The measure is part of an ongoing BARI research project on gentrification in Boston.
- *subwaylines_p_odp*: MBTA Subway lines (2012)

Upload datasets to GeoNode

In this tutorial you will upload the shapefiles to GeoNode. Here is how you can upload one of the datasets. Just repeat the same process for all of the datasets.

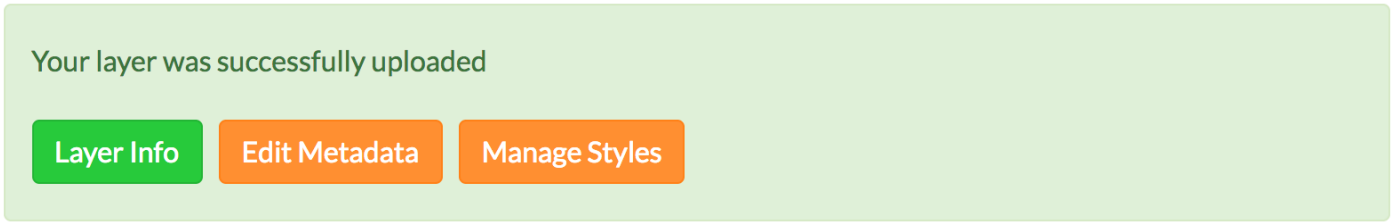
Click on the "Data > Upload Layer" menu. Then drag and drop the *biketrails_arc_p.zip* file to the "Drop files here" area, or use for this purpose the "Choose Files" button.



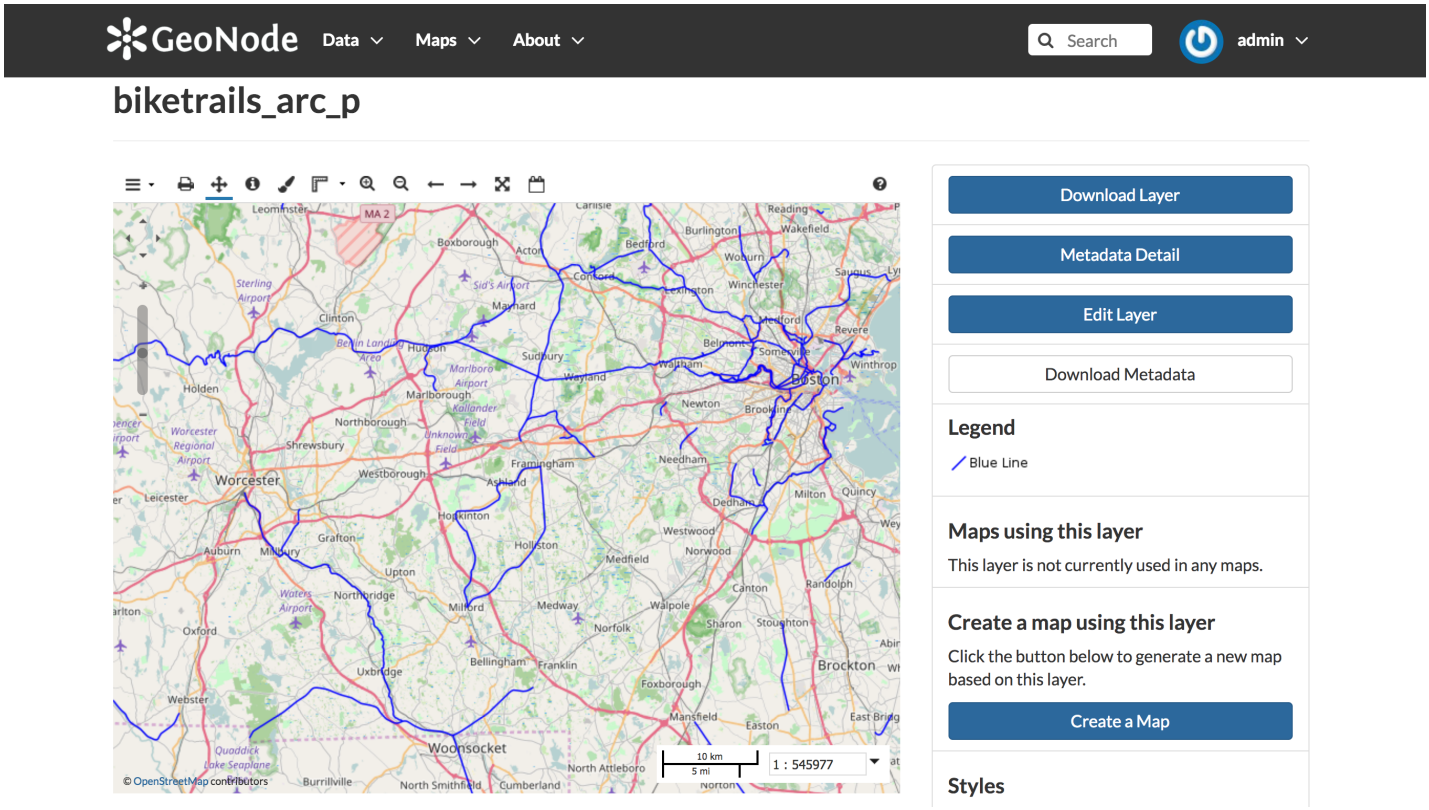
In a real use case scenario it would be possible to give granular permissions for this dataset to users or group of users. It is possible to do this using the "permissions" box.

For the purpose of this workshop we will leave default permissions for each datasets: this means that anyone will be able to see and download the datasets, but only the data owner and GeoNode administrators will be able to change the metadata, edit the styles, and administer the dataset.

Click on the "Upload Files" button. If everything works smoothly, you should see the following info box at the end of the upload process:



Now click on the "Layer Info" button, and you will get to the layer page.



Using the map widget you can browse the layer, use the "Identify" tool to query it, change the base maps and access to the style editor.

Metadata edit

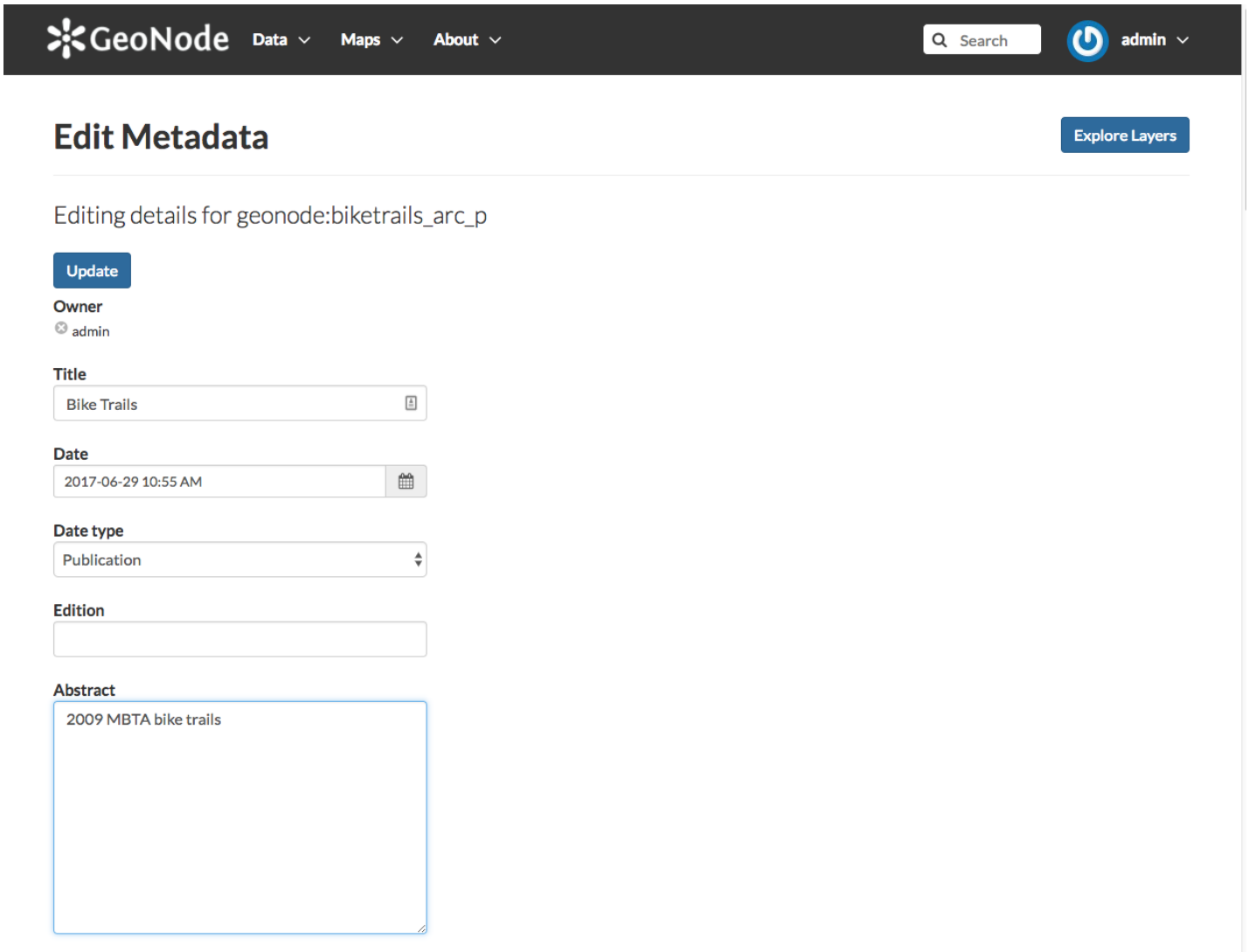
Now, you are going to edit some of the layer's metadata.

From the layer page, click on "Edit Layer" button, and then on "Edit Metadata" button.

Edit the *biketrails_arc_p* layer using the following information:

- Title: "Bike Trails"
- Abstract: "2009 MBTA bike trails"

- Regions: “United States of America”
- Keywords: “boston, FOSS4G2017, Commute”
- Category: “Transportation”



Repeat the process for the other datasets

Repeat the upload process for each shapefile, and compile the metadata using this information:

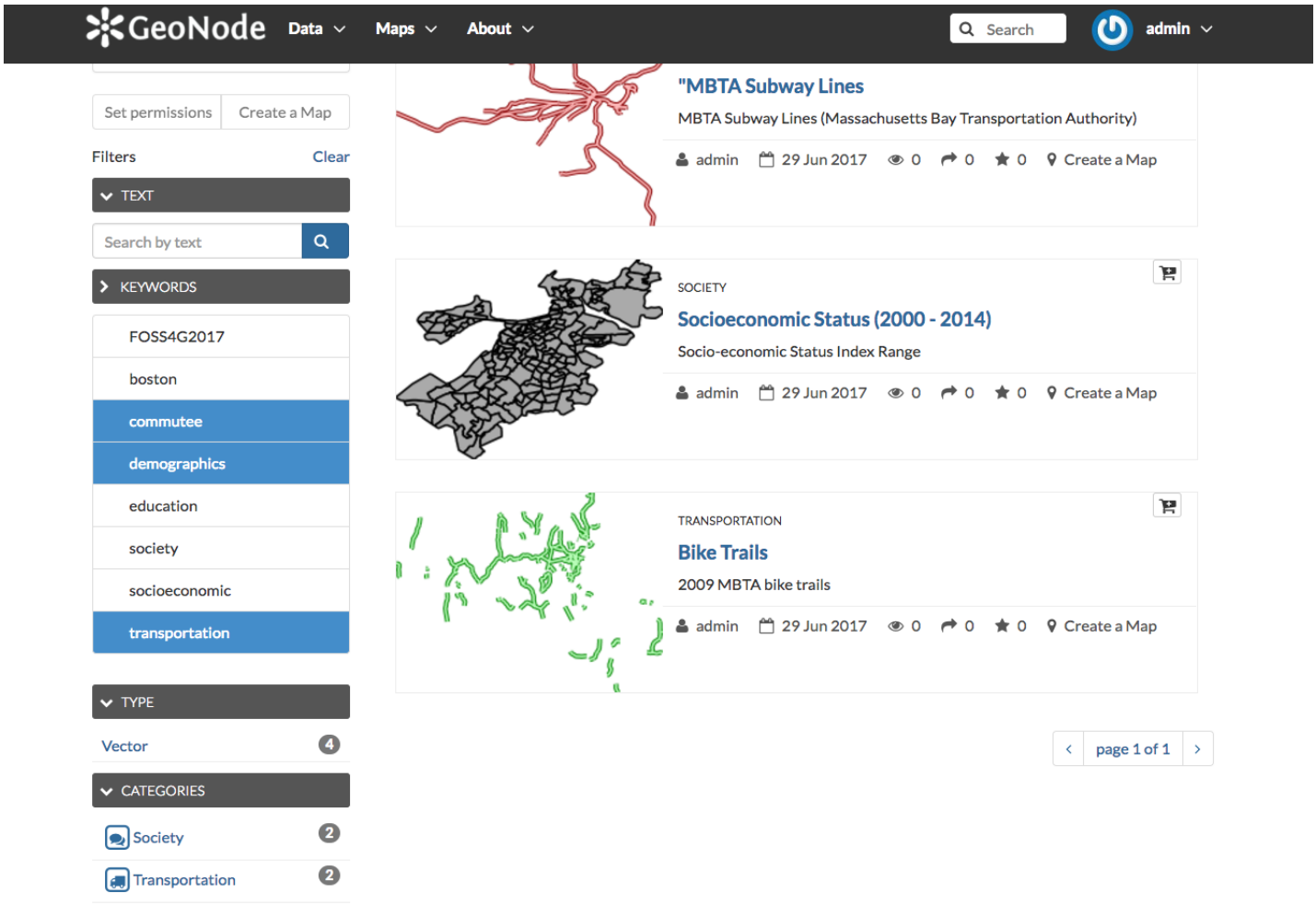
- boston_public_schools_2012_z1l
  - Title: “Boston Public Schools (2012)”
  - Abstract: “All schools in the Boston Public Schools District (2012)”
  - Regions: “United States of America”
  - Keywords: “boston, FOSS4G2017, education, society”
  - Category: “Society”
- socioeconomic_status_2000_2014_9p1

    - Title: “Socioeconomic Status (2000 - 2014)”
    - Abstract: “Socio-economic Status Index Range”
    - Regions: “United States of America”
    - Keywords: “boston, FOSS4G2017, society, demographics, socioeconomic”
    - Category: “Society”
- subwaylines_p_odp
    - Title: “MBTA Subway Lines”
    - Abstract: “MBTA Subway Lines (Massachusetts Bay Transportation Authority)”
    - Regions: “United States of America”
    - Keywords: “boston, FOSS4G2017, transportation”
    - Category: “Transportation”

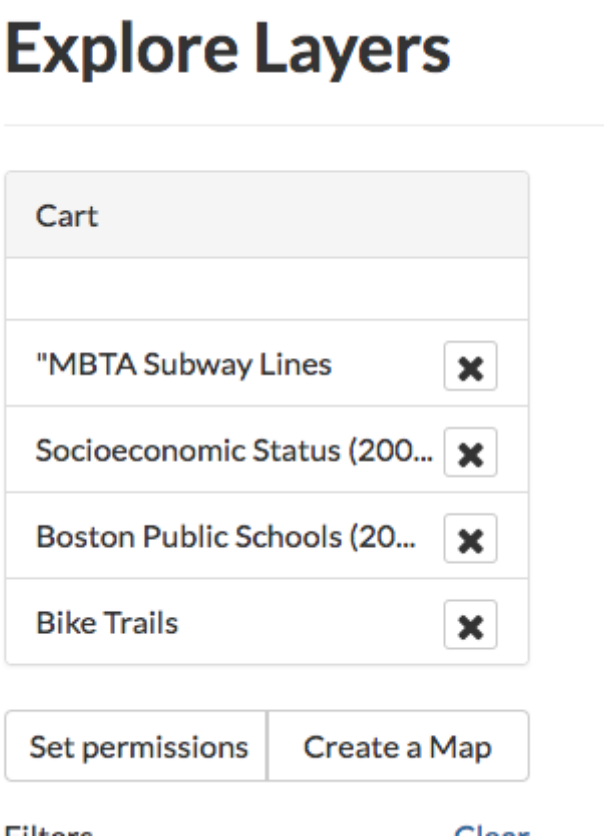
Create a map

Now go to the layers page (“Data > Layers”).

You should be able to see all of the layers. You should also be able to filter layers based on a text search, keywords, categories, regions and even date ranges.

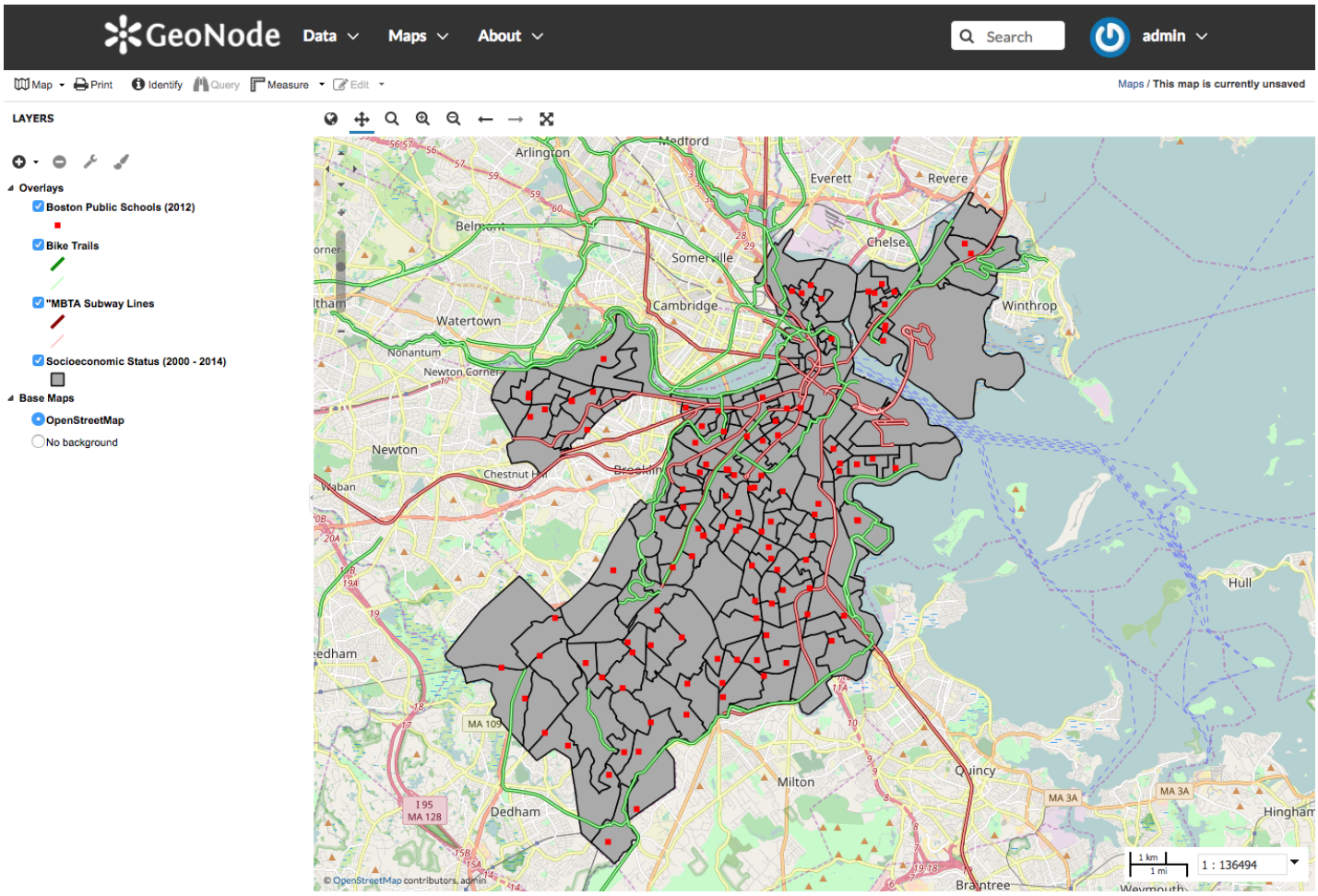


What you are going to do now it is to compose a map with these layers you just uploaded. For this purpose click on the cart icon at the right of each layer. They will be added in the cart area as in the following figure



At this point click on the “Create a Map” button: the map composer, which is based on GeoExplorer, should open with the layers in the map.

Center the map in the Boston area. Then drag and drop each layer in the table of contents area, and reorder all of the layers as in the following figure



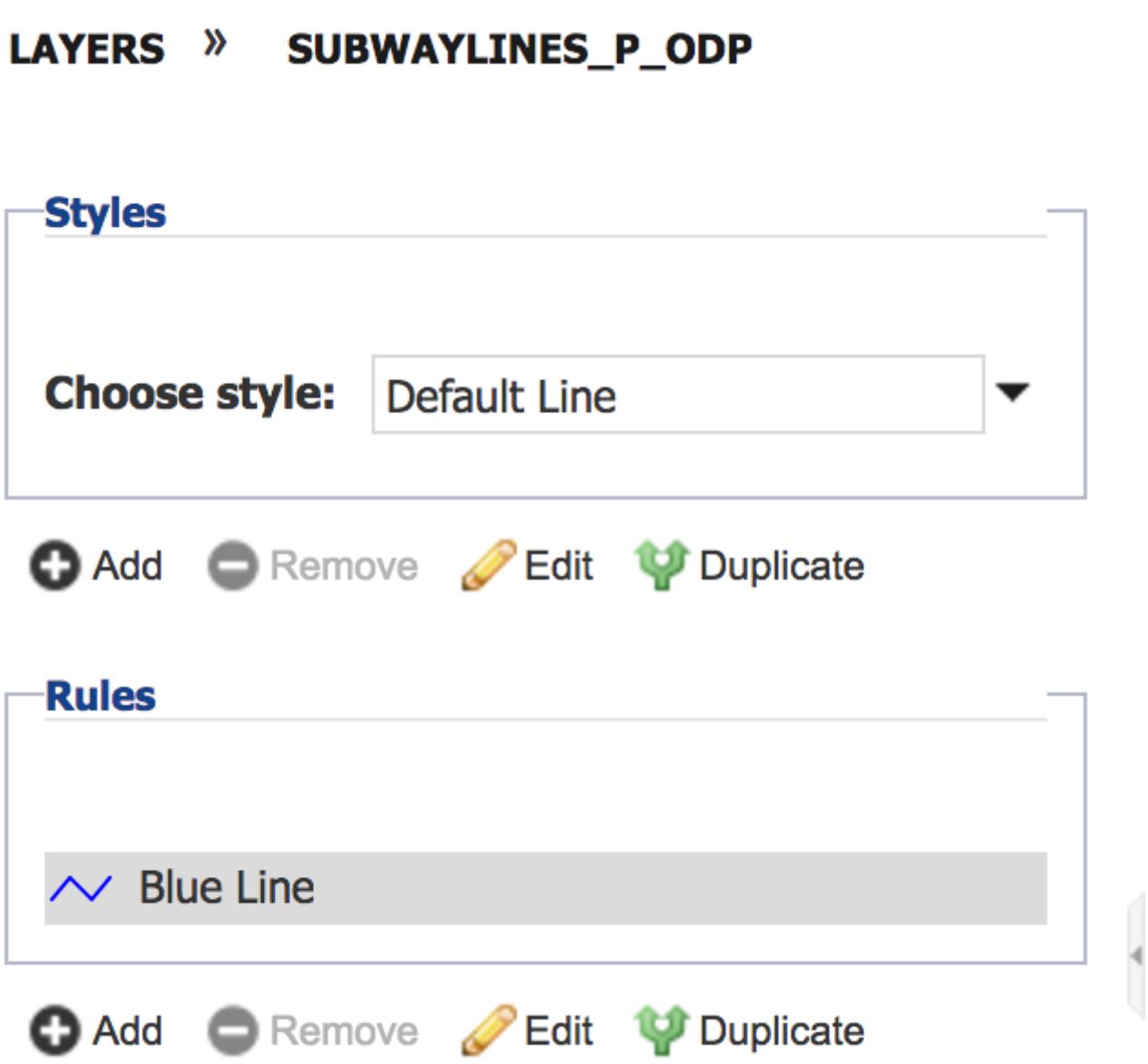
Edit the style of the layers

Using the GeoNode map composer it is possible to edit layer styles. Let's change the style for the "MBTA Subway Lines" layer.

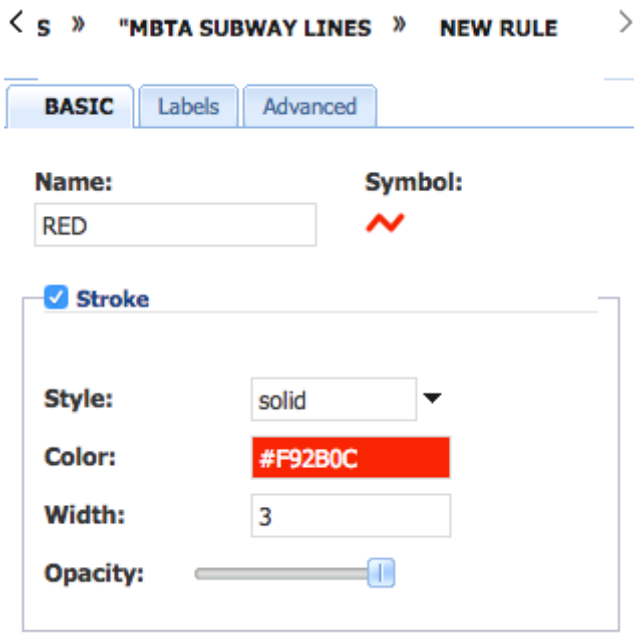
Right click on that layer, and press "Layer Styles". The Styles Editor widget will open.

For this layer, we will create one rule for each Boston Subway Line (Blue, Green, Orange, Red, Silver). The "MBTA Subway" layer has an attribute, named "LINE", which is set to "BLUE", "GREEN", "ORANGE", "RED", "SILVER", and that can therefore be used for this purpose.

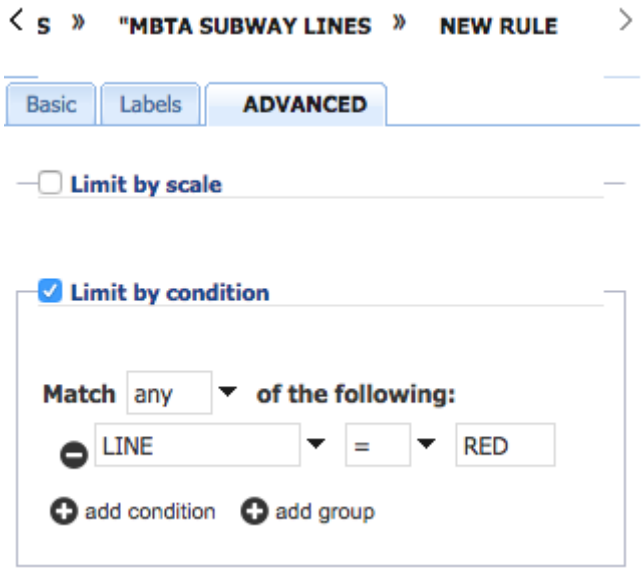
Select the default rule (named "Blue Line" in the figure), and press the "Edit" button under it.



In the “Basic” tab, type “RED” in the “Name” textbox and select a Red color. As a width value, type “5”.



In the “Advanced” tab, check the “Limit by condition” checkbox. Then add the following condition: “LINE = RED”, and save it.

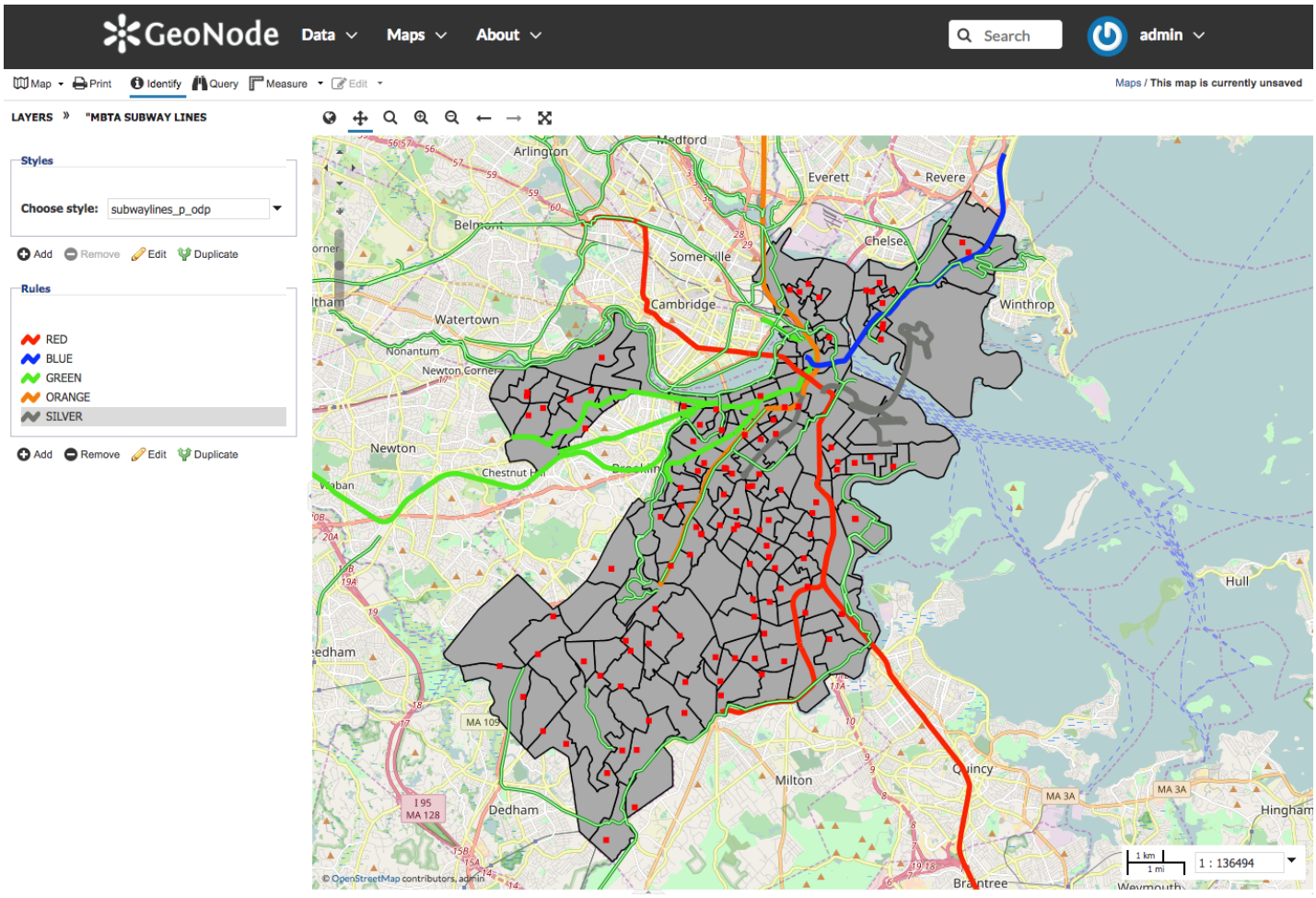


Repeat the process for the other lines. To add new rules to the default one, click on the “Add” button under the “Rules” area.

Those are all the rules which you will need to add:

- Rule name: “RED”, Condition: “LINE = RED”
- Rule name: “GREEN”, Condition: “LINE = GREEN”
- Rule name: “ORANGE”, Condition: “LINE = ORANGE”
- Rule name: “BLUE”, Condition: “LINE = BLUE”
- Rule name: “SILVER”, Condition: “LINE = SILVER”

This is how the layer style should look at the end:



Now, using the same approach based on rules, edit the style for the "Socioeconomic Status (2000-2014)".

Create three different rules, using the attribute *SESdx14*:

Rule 1:

- Name: "Low"
- Fill Color: Red
- Match: Any of the condition
- Condition 1: "SESdx14 >= -6"
- Condition 2: "SESdx14 < -2"

Rule 2:

- Name: "Mid"
- Fill Color: Yellow
- Match: Any of the condition

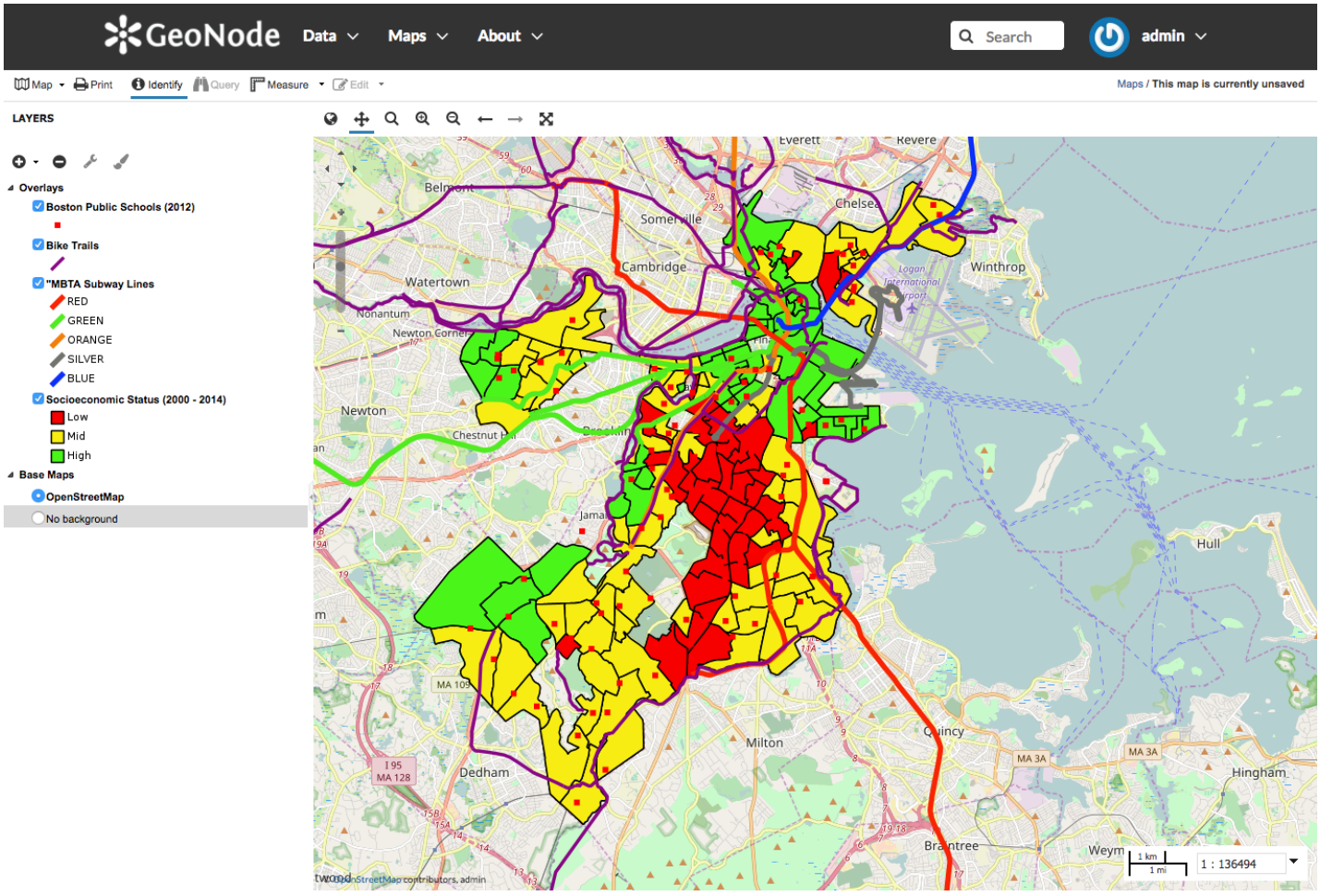
- Condition 1: “SESdx14 >= -2”
- Condition 2: “SESdx14 < 1”

Rule 3:

- Name: “High”
- Fill Color: Green
- Match: Any of the condition
- Condition 1: “SESdx14 >= 1”

Finally for the layer “Bike Trails” remove one of the default two rules. Make the other rule of a purple color.

Your map should look similar to this now:



Save the map

As a final step, do not forget to save the map by clicking on the “Map > Save Map” button.

Name the map something like “Boston Map at FOSS4G2017”

foss4g_2017_geonode_solr

Programming GeoNode with Python

GeoNode is built on top of Django, a really popular and powerful Python web framework.

> Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

You can have a great introduction to Django using the official Django tutorial.

> Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy which emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax which allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

> Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

In case you are new to Python, some very good resourced to get introduced to Python are:

- A bite of Python
- Google's Python Class
- Learn Python the hard way

In this step of the workshop you will use Python to interact with GeoNode.

Start the Django shell

A great way to start using GeoNode with Python is by using the Django shell.

In order to open it you will need to run the Django manage.py command:

```
$ cd /workshop/geonode/
python manage.py shell
```

You could receive an error like this:

```
$ python manage.py shell
Traceback (most recent call last):
  File "manage.py", line 27, in <module>
    from django.core.management import execute_from_command_line
ImportError: No module named django.core.management
```

If you receive this error message it is because you forgot to activate the virtualenv. To activate it you can do this:

```
$ . /workshop/env/bin/activate
```

If everything went smoothly the Django shell, based on IPython, should correctly open:

```
$ python manage.py shell
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
Type "copyright", "credits" or "license" for more information.

IPython 5.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Play with the Django shell

The Django shell is a very useful tool to interact with GeoNode using a programmatic approach based on Python.

Without providing too many details, GeoNode is based on three main models:

- *Layer*, a model representing a GeoNode layer
- *Map*, a model representing a GeoNode map
- *Document*, a model representing a GeoNode document

By using the GeoNode library it is possible to access to real instances of layers, maps and documents. After getting one instance it will be possible to read or even set properties of that instance.

Start trying same sample code using the Django shell. As a first step you need to import the Layer and Map models:

```
>>> from geonode.layers.models import Layer
>>> from geonode.maps.models import Map
```

Now try to access one of the layers in your instance, and print in the terminal some of the properties:

```
>>> layer = Layer.objects.all()[0] # get the layer
>>> print layer.title # get the layer's title
Bike Trails
>>> print layer.abstract # get the layer's abstract
2009 MBTA bike trails
>>> print layer.keywords.all() # get the layer's keywords
[<HierarchicalKeyword: FOSS4G2017>, <HierarchicalKeyword: commutee>]
>>> print layer.category # get the layer's category
Transportation
>>> print layer.region_name_list() # get the layer's regions
[u'United States of America']
>>> print layer.bbox # get the layer's bounding box
[Decimal('-73.4114198360'), Decimal('41.3945932051'), Decimal('-69.9476115692'), Deci
>>> print layer.get_thumbnail_url() # get the layer's thumbnail
http://localhost:8000/uploaded/thumbs/layer-cda5c5ae-5cfa-11e7-8103-02d8e4477a33-thum
>>> print layer.owner, layer.metadata_author, layer.poc # get the layer's owner, meta
admin admin admin
```

In the next code block, you iterate the layer's attributes (fields) to get their name, type and label:

```
>>> for attribute in layer.attributes:
>>>     print attribute.attribute, attribute.attribute_type, attribute.attribute_labe

TRAIL_STAT xsd:string Status of the trail
OWNER xsd:string Owner
PREV_OWNER xsd:string Previous Owner
MANAGER xsd:string Manager
STATUS_OWN xsd:string Status of owner
STATUS_MAN xsd:string Status of manager
TRAILNAME xsd:string Name of the trail
...
```

Using the same approach you followed for the layer, you can programmatically get to the map you previously created:

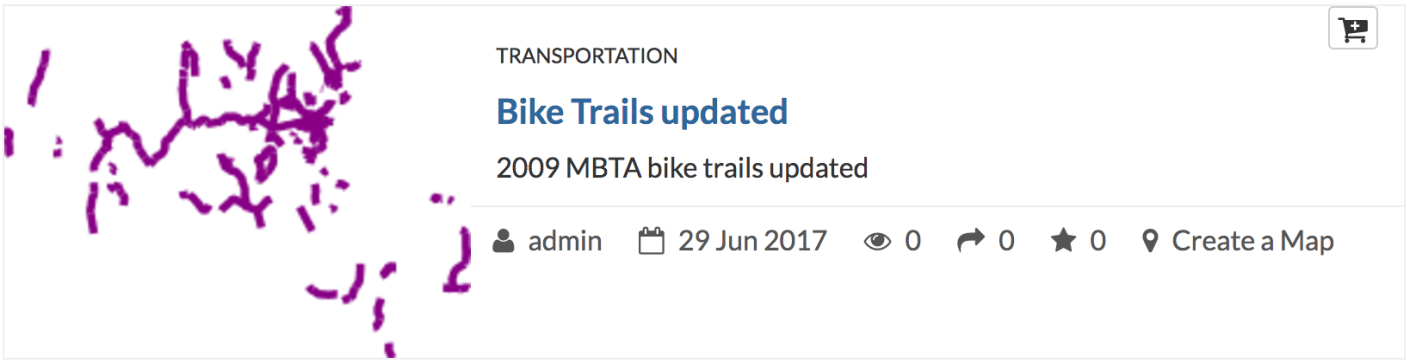
```
>>> map = Map.objects.all()[0] # get the map
>>> print map.title # get the map's title
Boston Map at FOSS4G2017
>>> print map.abstract # get the map's abstract
A Boston map created for the FOSS4G 2017 GeoNode and Solr Workshop
>>> print map.bbox # get the map's bounding box
[Decimal('-71.2196746791'), Decimal('42.3222257127'), Decimal('-70.8763491837'), Deci
>>> print map.get_thumbnail_url() # get the map's thumbnail url
http://localhost:8000/uploaded/thumbs/map-3b90f64a-5d11-11e7-8103-02d8e4477a33-thumb.
>>> print map.owner, map.metadata_author, map.poc # get the map's owner, metadata aut
admin admin admin
>>> print len(map.layers) # get the map's number of layers (this count includes basem
6
```

Using Python to update layer's metadata

The Django shell can be used to programmatically update the metadata of a layer or a map. Here is what you can do to change the title and the abstract of a layer:

```
>>> layer = Layer.objects.all()[0]
>>> layer.title = 'Bike Trails updated'
>>> layer.abstract = '2009 MBTA bike trails updated'
>>> layer.save()
```

If now you open the layers page, you should see the metadata for the title and abstract correctly updated:



As a bonus step, try to change the title and abstract of the map you created.

Using Python to read layer's features

GeoNode comes with GDAL and its Python bindings installed. GDAL is a great toolset and library to interact with a plethora of geospatial formats.

> The Geospatial Data Abstraction Library (GDAL) is a computer software library for reading and writing raster and vector geospatial data formats, and is released under the permissive X/MIT style free software license by the Open Source Geospatial Foundation. As a library, it presents a single abstract data model to the calling application for all supported formats. It may also be built with a variety of useful command line interface utilities for data translation and processing. Projections and transformations are supported by the PROJ.4 library.

> The related OGR library (OGR Simple Features Library), which is part of the GDAL source tree, provides a similar ability for simple features vector graphics data. GDAL was developed mainly by Frank Warmerdam until the release of version 1.3.2, when maintenance was officially transferred to the GDAL/OGR Project Management Committee under the Open Source Geospatial Foundation.

> GDAL/OGR is considered a major free software project for its "extensive capabilities of data exchange" and also in the commercial GIS community due to its widespread use and comprehensive set of functionalities.

Here you will use the GDAL Python bindings to read the "Boston Public Schools (2012)" shapefile store in GeoNode/GeoServer.

```
>>> from osgeo import ogr
>>> schools_shapefile = "/vagrant/geonode/geonode/uploaded/layers/boston_public_schoo
>>> driver = ogr.GetDriverByName('ESRI Shapefile')
>>> data_source = driver.Open(schools_shapefile, 0) # 0 means read only
>>> shp_layer = data_source.GetLayer() # shapefiles have just one layer
>>> for feature in shp_layer:
>>>   print feature.GetField("City"), feature.GetField("School_Nam"), feature.GetGeom

East Boston Adams Elementary POINT (238299.659967332147062 901795.889980231411755)
Brighton Another Course to College POINT (229148.81013232798432 900003.22998057480435
Brighton Baldwin ELC POINT (229615.22020976812928 899151.699935027398169)
...
```

You can check the shapefile data store path using the GeoServer administrative interface or the *locate* Linux command. If you didn't customize things it should be */vagrant/geonode/geonode/uploaded/layers/boston_public_schools_2012_z1l.shp*

A GDAL vector data store can be accessed using OGR. OGR has different drivers for different data formats. In your case the vectore dataset has been uploaded to GeoServer as a shapefile, therefore you can read it using the 'ESRI Shapefile' driver.

Each OGR dataset can be composed by different layers. Shapefiles are composed by just one layer.

In the previous code you are iterating all of the features of the default shapefile layer in order to print the feature's values for the "City" and "School_Nam" fields, and the WKT representation of the feature's geometry.

If you are interested in getting more information about the GDAL Python bindings, two great resources are:

- The Python GDAL/OGR Cookbook
- GeoProcessing with Python

NOTE: Another way to use GDAL with Python is with the Fiona libary. Fiona is often used with Shapely, which comes with GeoNode.

foss4g_2017_geonode_solr

OGC Services with GeoServer

Any SDI and geoportal needs to implement OGC Web Services (OWS) and specifications such as:

- OGC Web Map Service Standard, WMS
- OGC Web Feature Service Standard, WFS
- OGC Transactional Web Feature Service Standard, WFS-T
- OGC Web Coverage Service Standard, WCS
- OGC Web Processing Server Standard, WPS
- OGC Web Map Tile Service Standard, WMTS
- OSGeo Tile Map Service Specification, TMS
- OSGeo WMS Tile Caching Specification, WMS-C (*)
- OGC Catalogue Service, CSW

(*) superceded by OSGeo TMS and OGC WMTS

All of this services are provided in GeoNode by its underlying components: GeoServer, GeoWebCache and pycsw. It is possible to use GeoNetwork in place of pycsw, in case it is needed, by changing the GeoNode default configuration.

GeoServer implements WMS, WFS, WCS, WPS. GeoWebCache implements all of the cache map tiles services: TMS, WMS-C, WMTS. pycsw, or GeoNetwork, implements CSW.

GeoNode, in its end user standard interface, right now makes use of just WMS, WFS, WFS-T, WMS-C and CSW.

WMS, WFS and WFS-T services are provided by GeoServer. WMS-C services are provided by GeoWebCache, as you will see in the next workshop's tutorial. CSW services are provided, as you will see in a following workshop's tutorial, by pycsw.

In this tutorial you will take a quick tour of the GeoServer administrative interface and you will have an overview of the GeoServer OGC services used by GeoNode: WMS and WFS. Then you will expore the GeoServer REST API, which provides to GeoNode the way to interact with GeoServer when it is not possible to use OGC services (OWS) standard calls.

You will also use OWSLib, a Python library for interacting with OWS, in order to use these service with the Python language.

Before starting, make sure GeoNode and GeoServer are up and running at: http://localhost:8000/ and http://localhost:8080/geoserver/ Make sure you are logged in both of them.

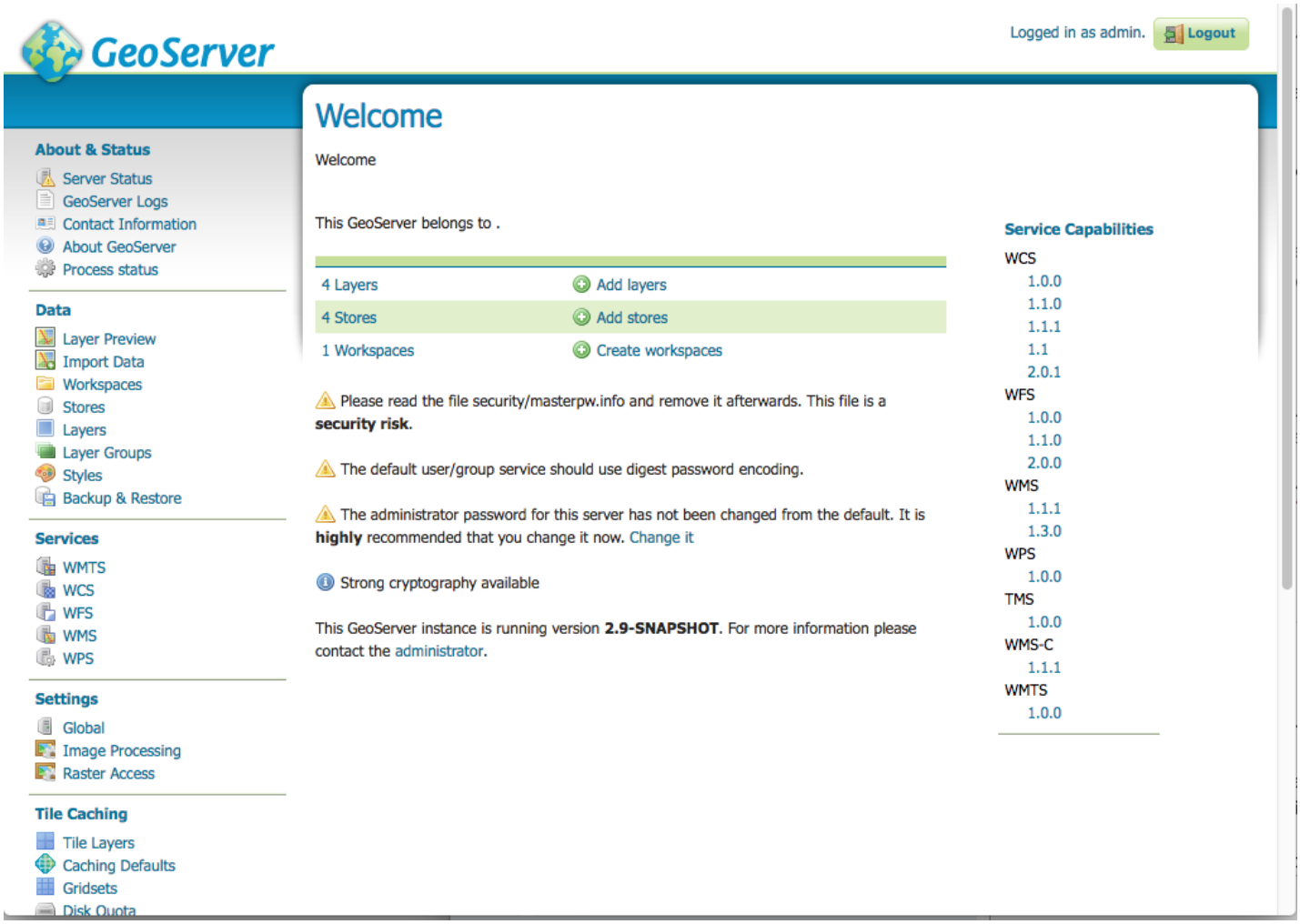
If GeoNode/GeoServer are not running, activate the virtualenv, start geoserver and run the Django development server:

```
$ cd /workshop/
$ . env/bin/activate
$ paver start_geoserver
$ ./manage.py runserver 0.0.0.0:8000
```

An overview of the GeoServer web administration interface

GeoServer has a browser-based web administration interface application used to configure all aspects of GeoServer, from adding and publishing data to changing service settings.

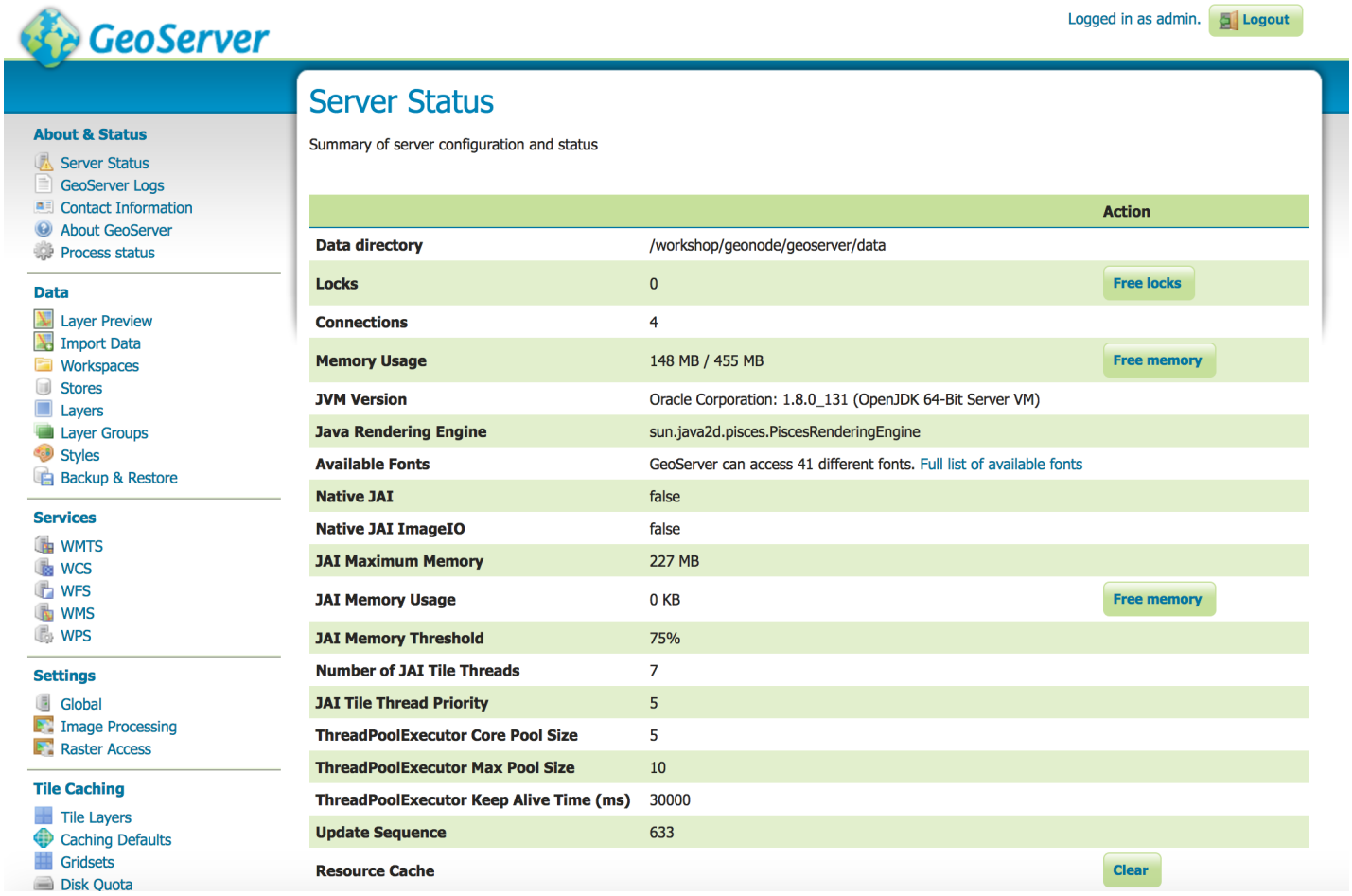
When accessing the home page of the GeoServer administrative site, using the url http://localhost:8080/geoserver (or from the GeoNode user menu), on the right side you will see a list of supported standard and specifications



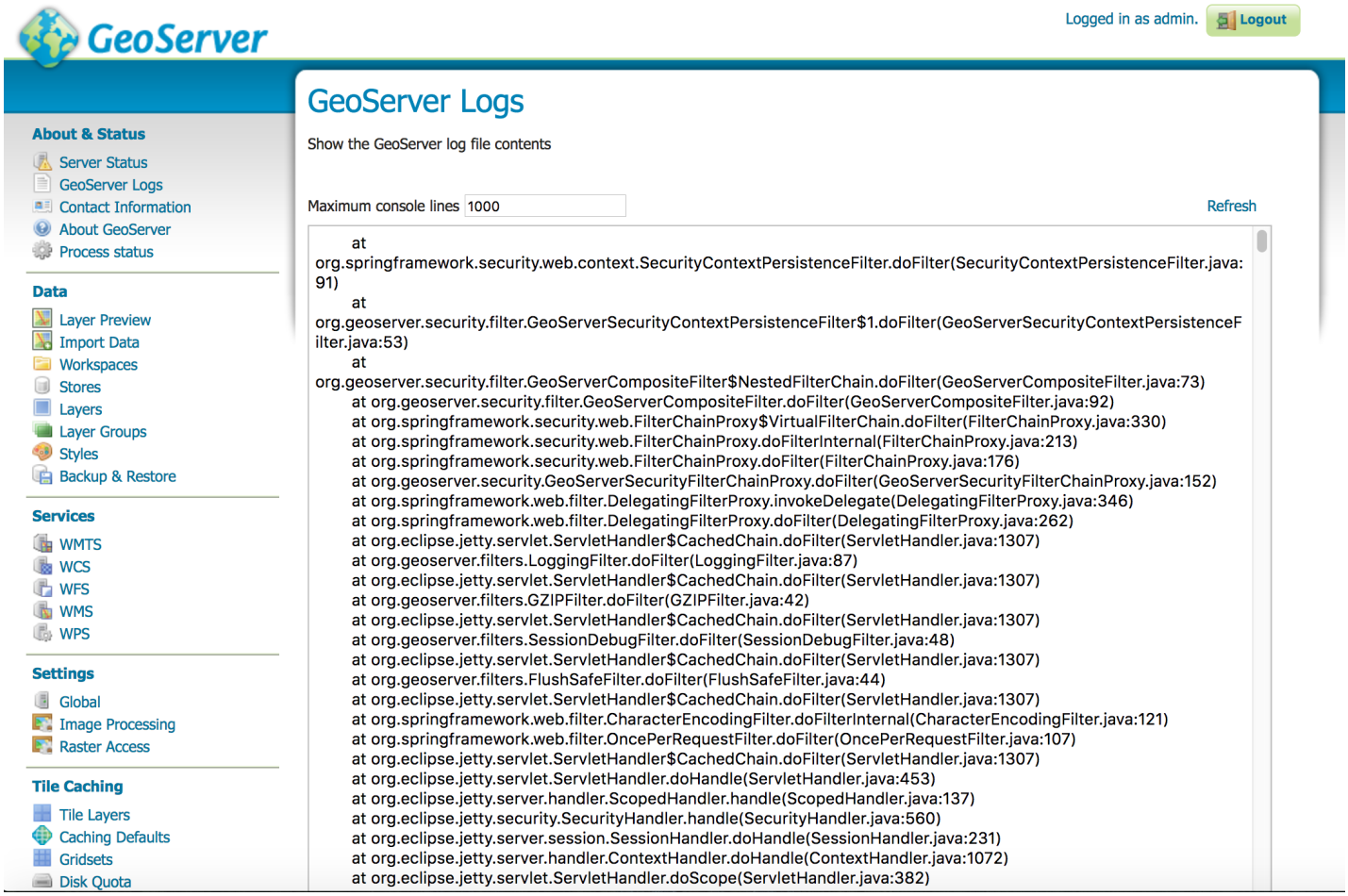
Clicking on each service capabilities link you will access the capabilities document for the given service type.

A quickstart for using the GeoServer web administration interface can be found here. For the purpose of the workshop, takes some minutes to explore the main sections of the GeoServer administrative interface which can be useful when using it with GeoNode. You can access to these sections by clicking on the menu items on the left of the GeoServer page.

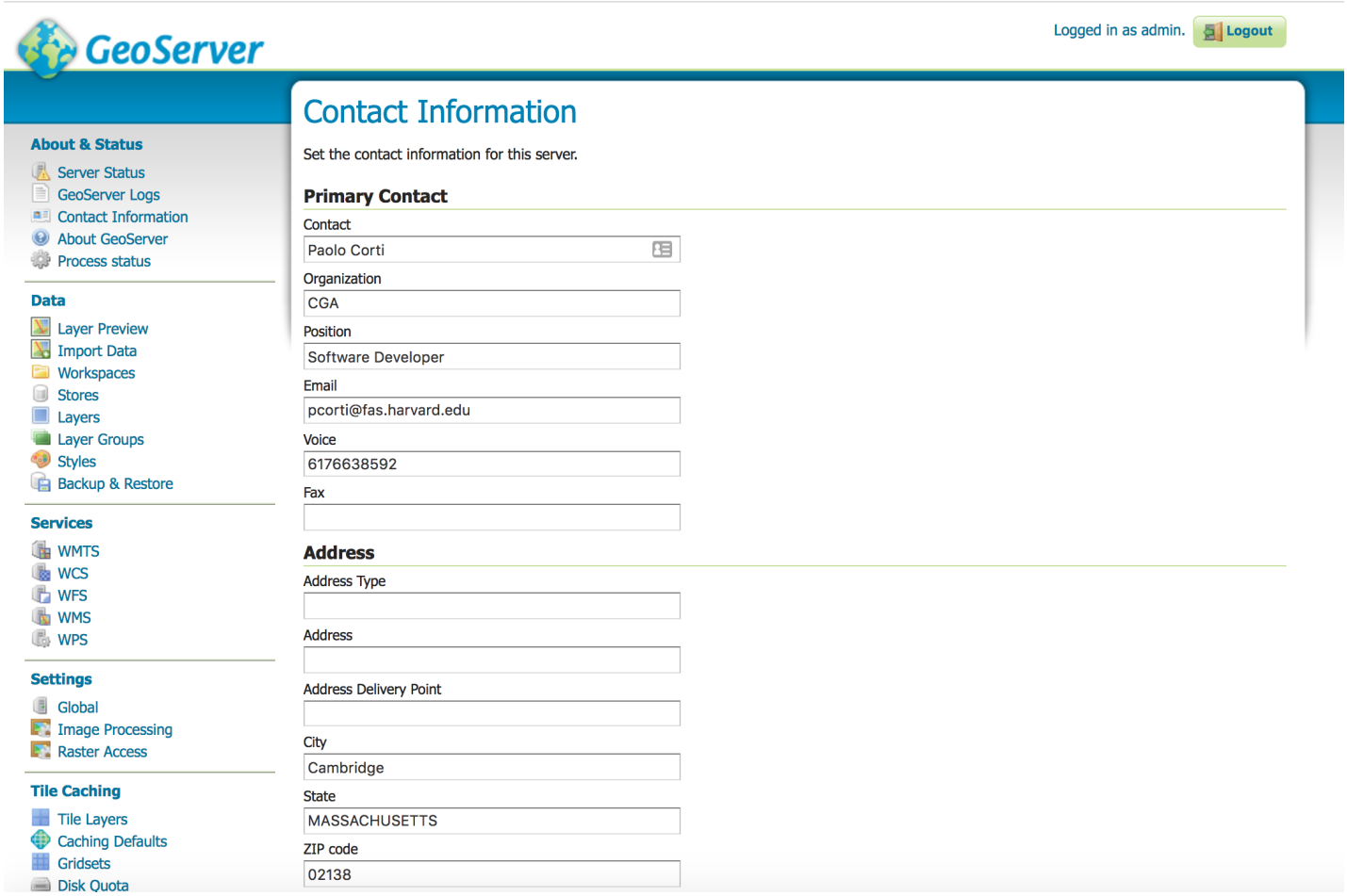
Server Status provides a summary of the server configuration and status. From this page it is possible to see the location of the GeoServer data directory, the connections and memory used, the JVM Version and many other information



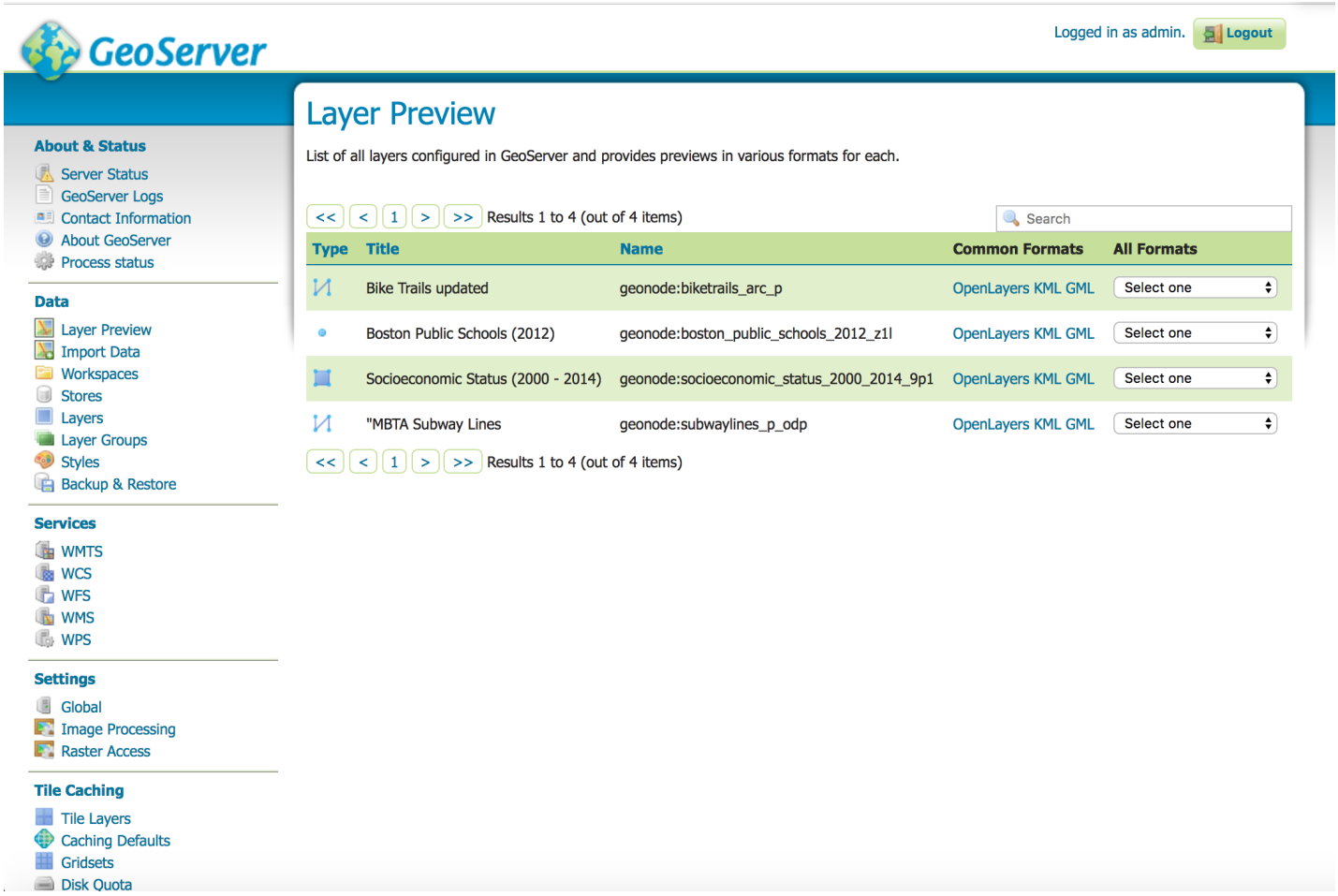
GeoServer Logs provides the log tail of the GeoServer servlet process. Obviously it is possible to access to it from the shell using linux commands such as *tail* and *less*

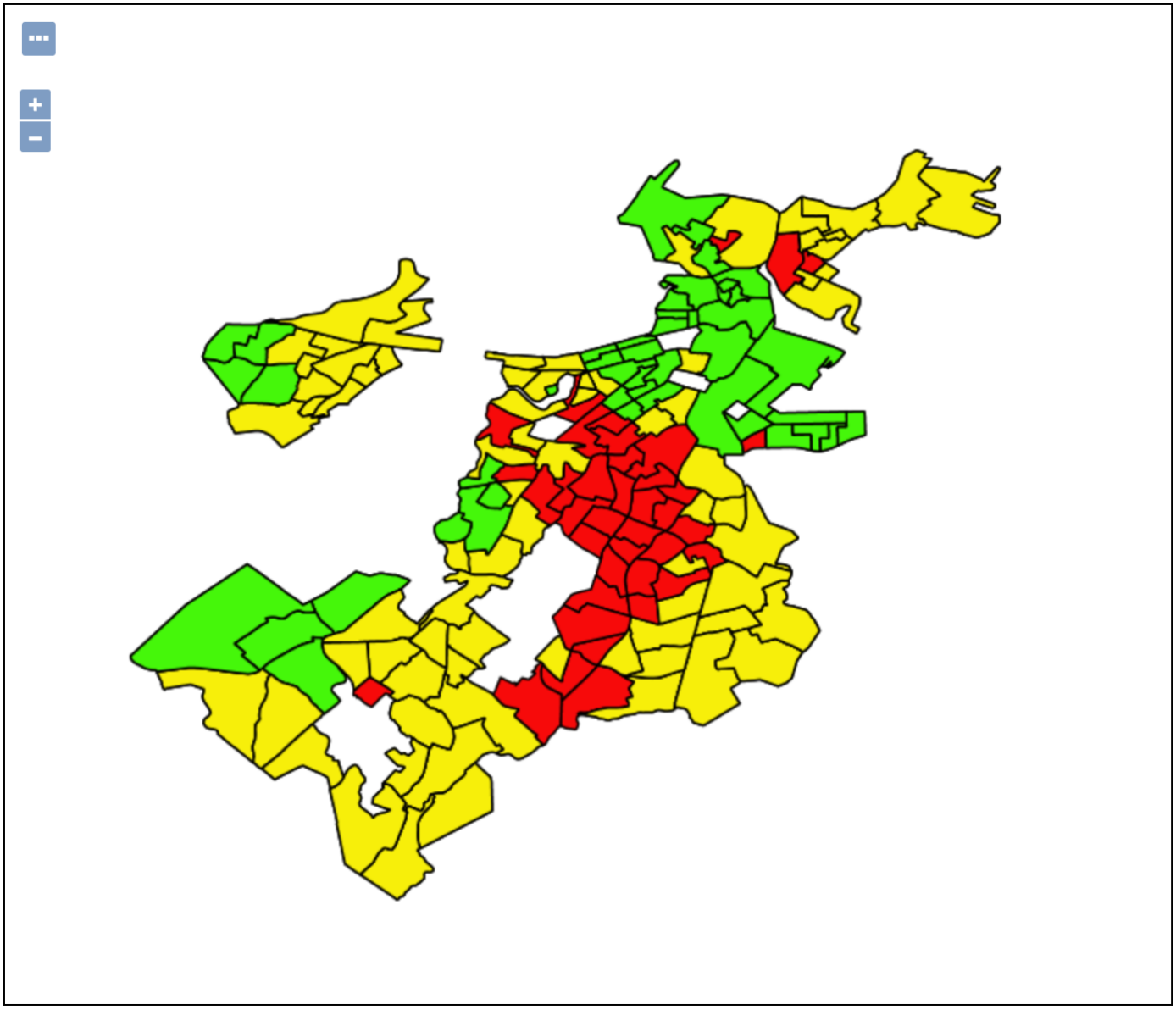


GeoServer Contact Information provides a form to set the contact information of the server. This information is used in the metadata provided by the WMS, WCS and WFS GetCapabilities documents.



Layer Preview provides a convenient interface for exploring the layers which have been uploaded to GeoNode/GeoServer. It is possible to browse layers using the OpenLayers viewer, or getting the WMS/WFS response in a specific format


GeoServer
Logged in as admin.
Logout
About & Status
Server Status
GeoServer Logs
Contact Information
About GeoServer
Process status
Data
Layer Preview
Import Data
Workspaces
Stores
Layers
Layer Groups
Styles
Backup & Restore
Services
WMTS
WCS
WFS
WMS
WPS
Settings
Global
Image Processing
Raster Access
Tile Caching
Tile Layers
Caching Defaults
Gridsets
Disk Quota
Layer Preview
List of all layers configured in GeoServer and provides previews in various formats for each.
<< < 1 > >> Results 1 to 4 (out of 4 items)
Search
Type
Title
Name
Common Formats
All Formats
Bike Trails updated
geonode:biketrails_arc_p
OpenLayers KML GML
Select one
Boston Public Schools (2012)
geonode:boston_public_schools_2012_z1l
OpenLayers KML GML
Select one
Socioeconomic Status (2000 - 2014)
geonode:socioeconomic_status_2000_2014_9p1
OpenLayers KML GML
Select one
"MBTA Subway Lines
geonode:subwaylines_p_odp
OpenLayers KML GML
Select one
<< < 1 > >> Results 1 to 4 (out of 4 items)

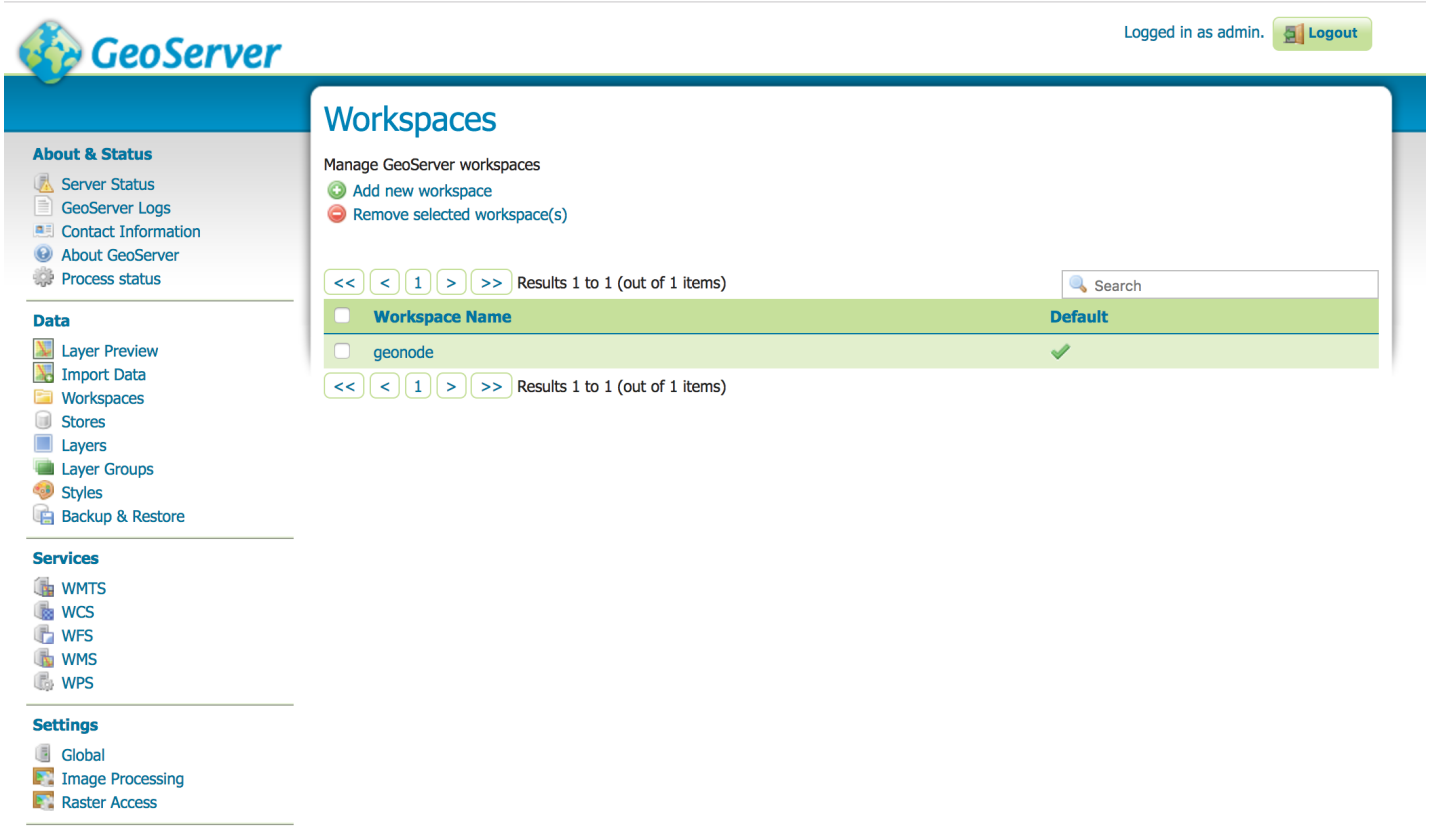


Scale = 1 : 136K

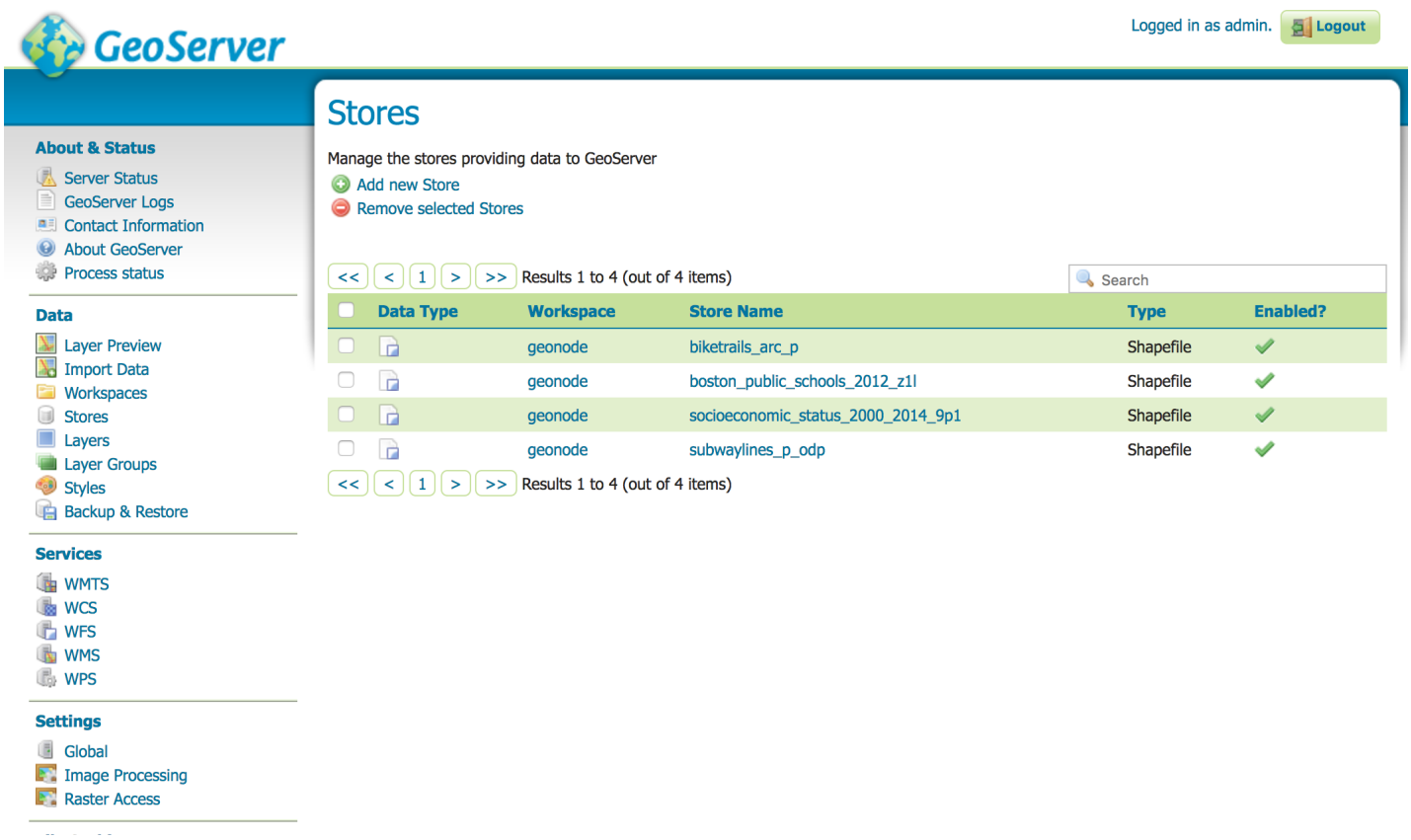
socioeconomic_status_2000_2014_9p1

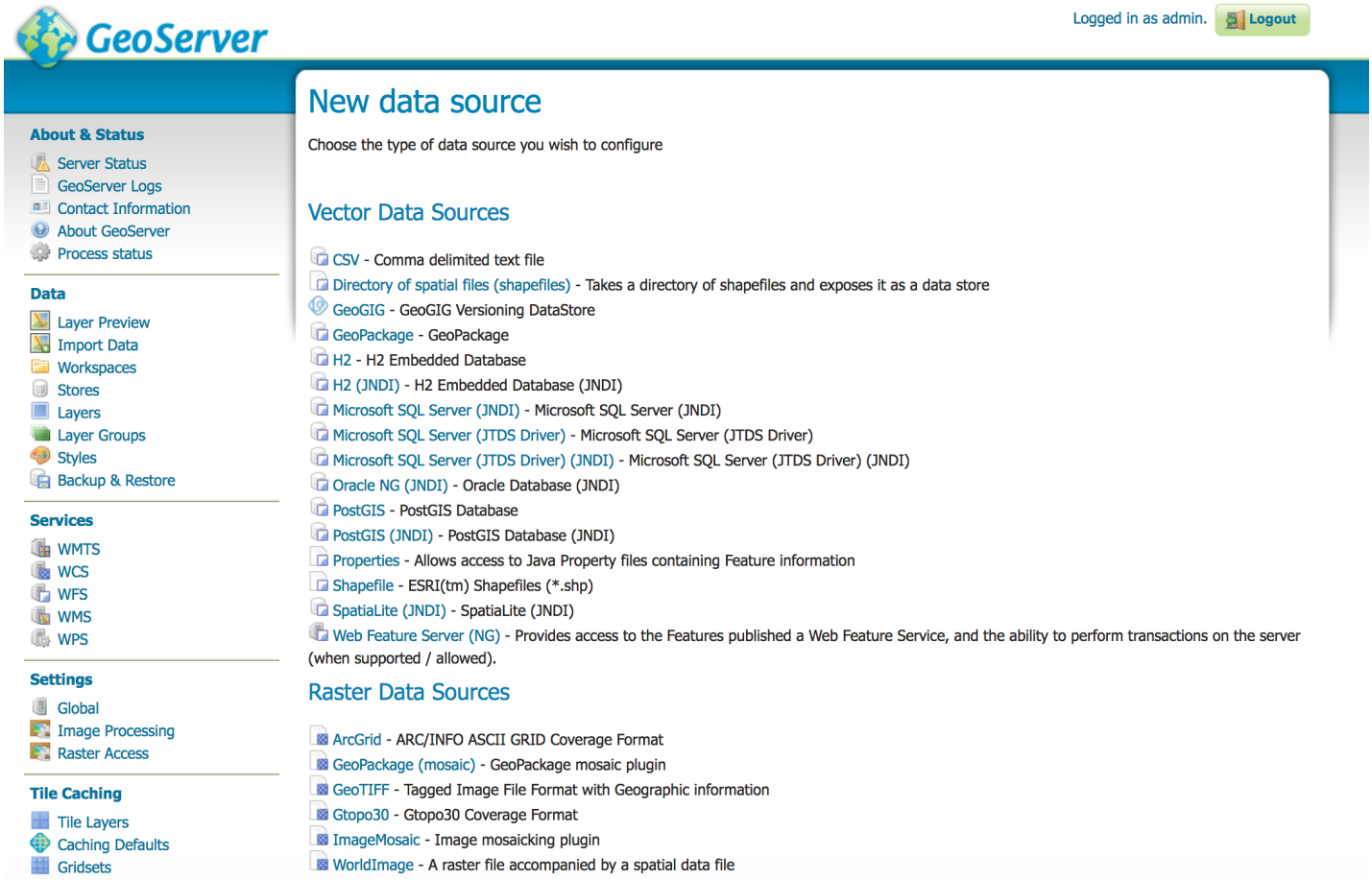
fid	CT_ID_1	GEOID10	ALAND10	AWATER1	POP100	HU100	Type	Res	BRA_PD_	BRA_
socioeconomic_status_2000_2014_9p1.78	25025090300	25025090300	380394	0	3179.0	1268.0	R	1.0	10	Roxbu

Workspaces Analogous to a namespace, a workspace is a container which organizes other items. In GeoServer, a workspace is often used to group similar layers together. By default GeoNode use a workspace named *geonode*, and all of the stores and layers are created within that workspace



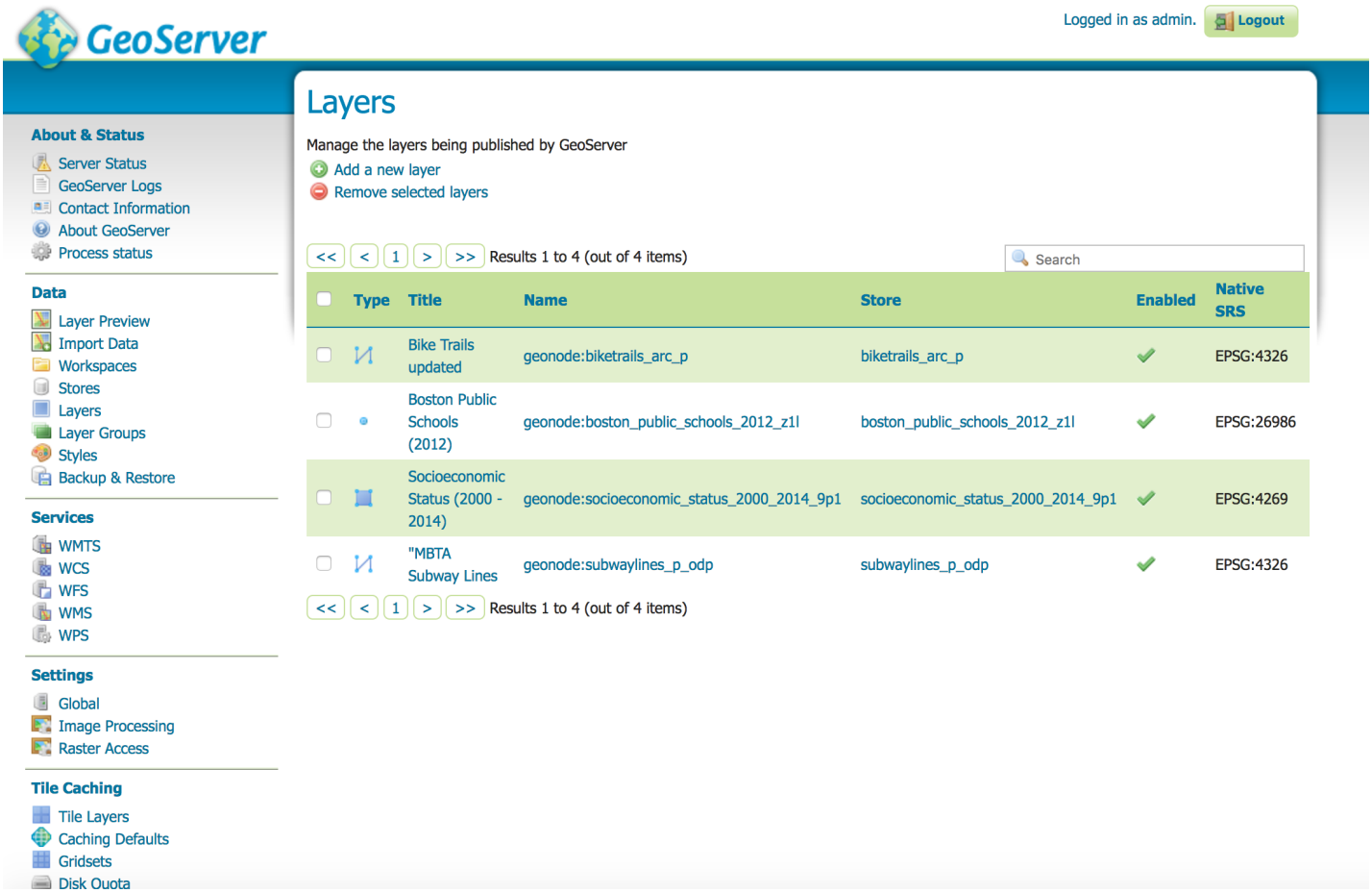
Stores A store connects to a data source that contains raster or vector data. A data source can be a file or group of files, a table in a database, a single raster file, or a directory. GeoNode by default will create a GeoTIFF store for each uploaded raster dataset and a Shapefile store for each uploaded vector dataset. It is possible to configure GeoNode to use one or more PostGIS databases and have the vector dataset uploaded as a table in the database. Furthermore, it is possible to use in GeoNode any datasource supported by GeoServer by registering the GeoServer layer in GeoNode with the *updatelayers* management command



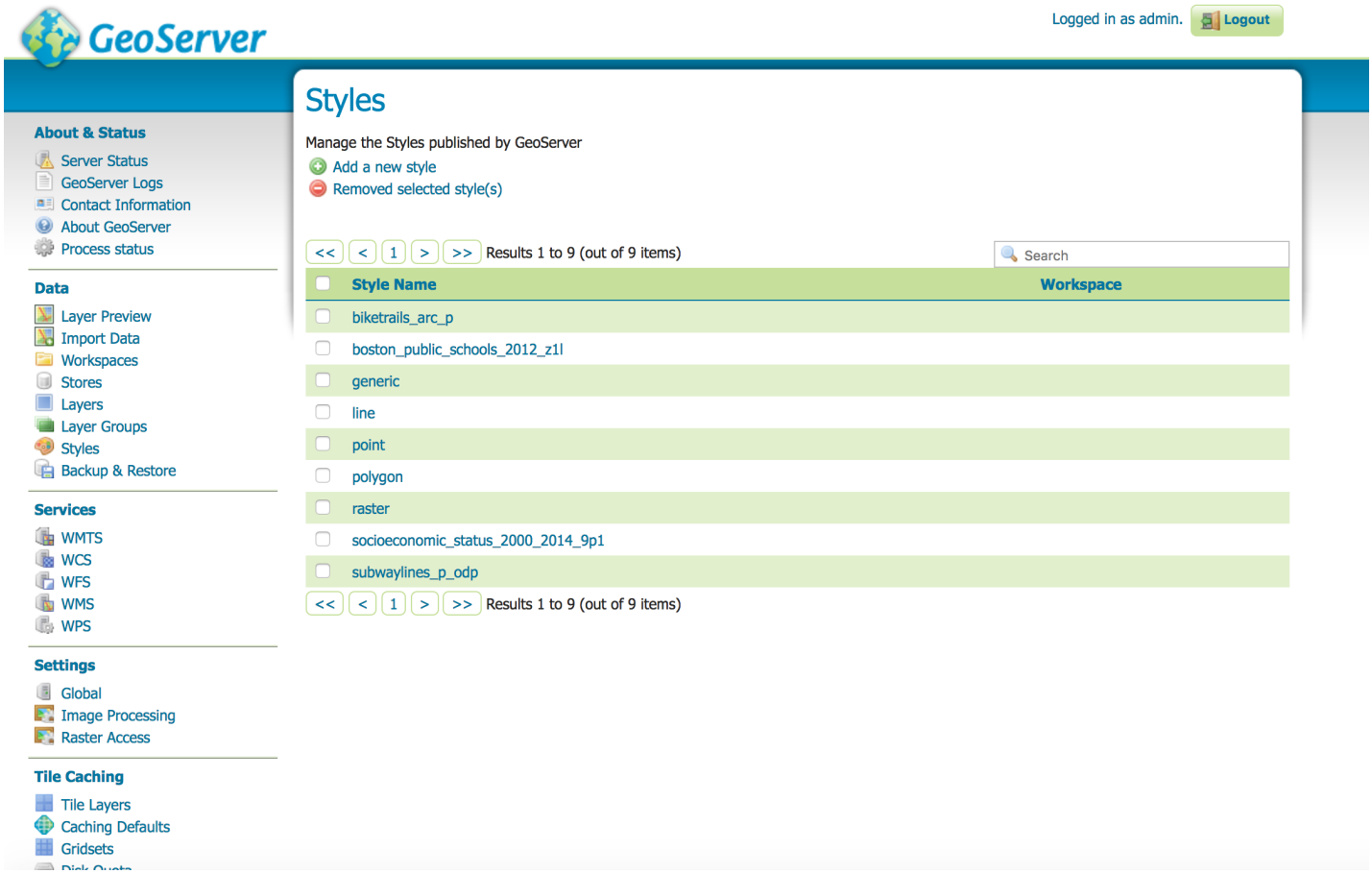


Layers provides a view of all of the layers contained in the GeoServer data directory. Layers can be raster or vector datasets. Each layer has a source of data, the *store*, which is

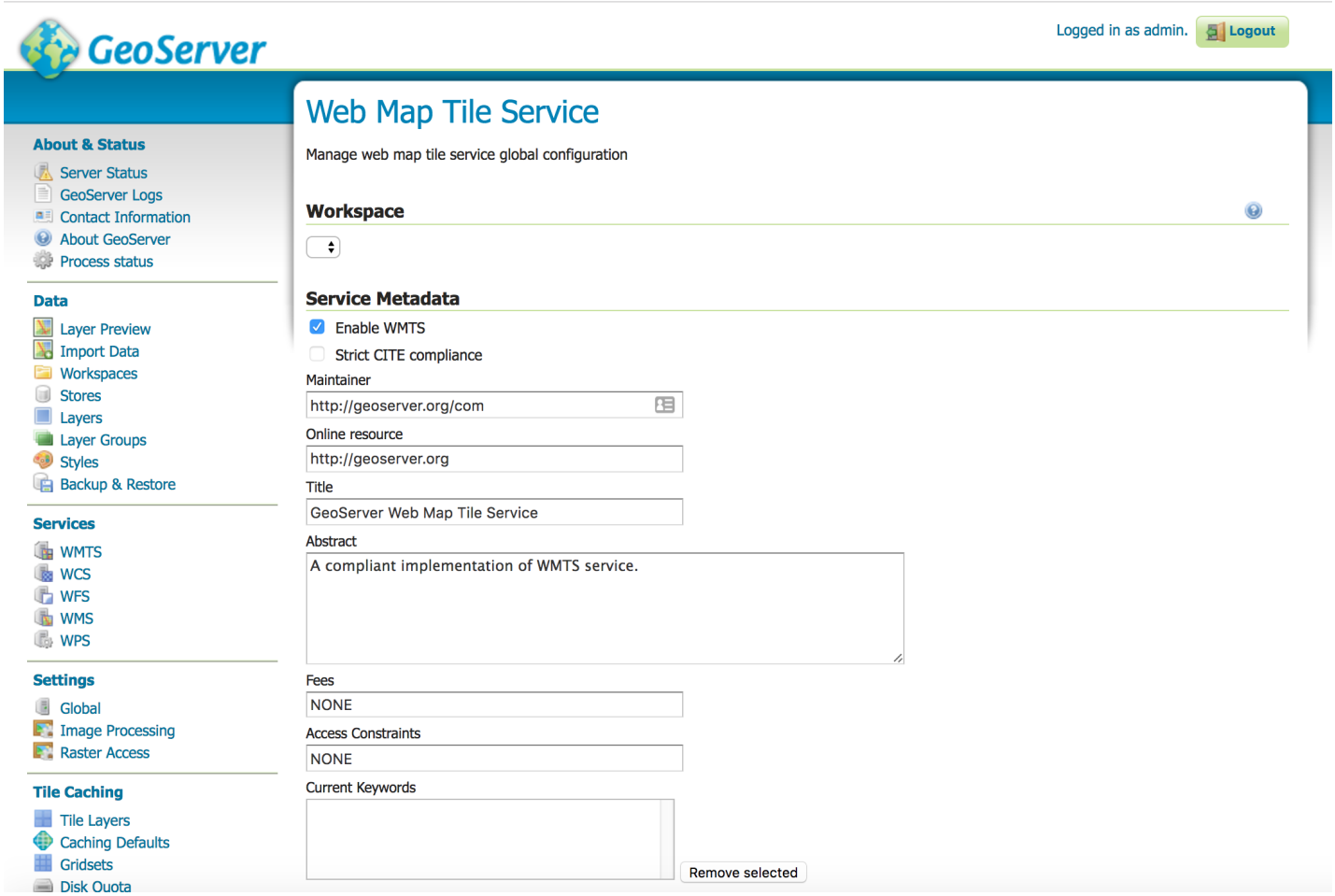
associated with the *workspace* in which the store is defined. As you can see the layers you have uploaded to GeoNode are all part of the *geonode namespace*, and for each one a shapefile store was created



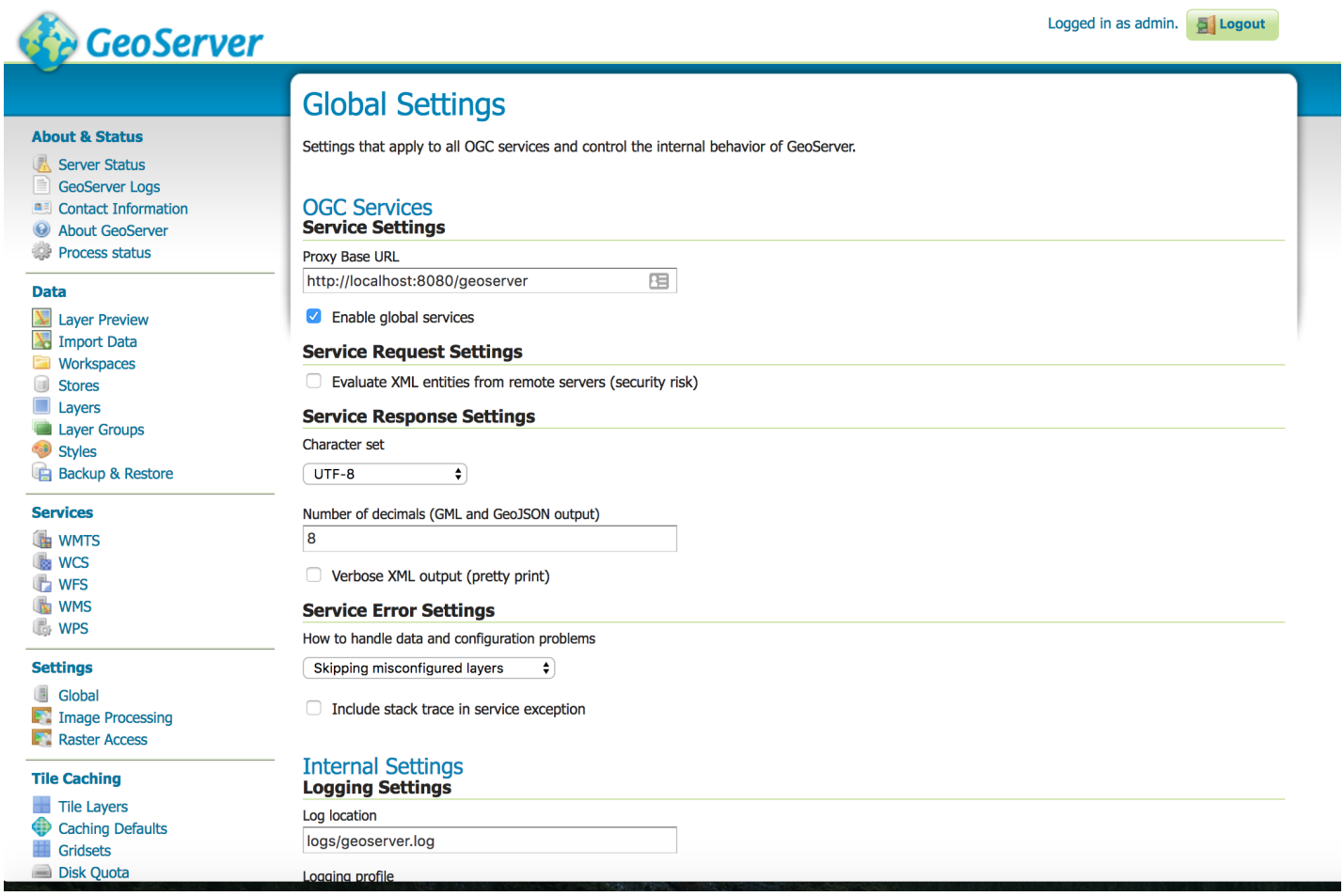
Styles provides a view of all of the styles associated (or not) to the layers. By default there is at least one style associated to a layer, which is the default style. One layer can have multiple styles associated to it. Styles can be created using the GeoExplorer interface of GeoNode or the GeoServer admin interface



WMTS, **WCS**, **WFS**, **WMS**, **WPS** provide forms to set the services metadata such as the maintainer, title, abstract and other info. One important thing that can be done from this forms is to enable/disable these services at global level (ie: for all of the layers).



Global Settings provides a way to alter global GeoServer settings, such as the logging level, the behavior with misconfigured layers, the number of decimals to include in output and many others



Tile Caching you will be back to this in the next workshop tutorial, when you will have a look at GeoWebCache.

Security when using GeoServer with GeoNode you shouldn't need to interact with this part of the GeoServer administrative site.

A gentle introduction to the WMS and WFS OGC standards

In this step of the tutorial you will explore the WMS and WFS OGC standards provided by GeoServer.

WMS

> The OGC Web Map Service (WMS) specification defines an HTTP interface for requesting georeferenced map images from a server. GeoServer supports WMS 1.1.1, the most widely used version of WMS, as well as WMS 1.3.0.

> WMS provides a standard interface for requesting a geospatial map image. The benefit of this is that WMS clients can request images from multiple WMS servers, and then combine them into a single view for the user. The standard guarantees that these images

> can all be overlaid on one another as they actually would be in reality. Numerous servers and clients support WMS.

For an extended reference to the WMS standard in GeoServer check the GeoServer WMS reference documentation

GeoServer implements the following WMS Operations (the last three ones are optional for this standard, but implemented in GeoServer):

- **GetCapabilities**: Retrieves metadata about the service, including supported operations and parameters, and a list of the available layers
- **GetMap**: Retrieves a map image for a specified area and content
- **GetFeatureInfo**: Retrieves the underlying data, including geometry and attribute values, for a pixel location on a map
- **DescribeLayer**: Indicates the WFS or WCS to retrieve additional information about the layer
- **GetLegendGraphic**: Retrieves a generated legend for a map
- **GetStyles**: Retrieves a list of styles associated to a layer

All of these WMS operations are used in GeoNode, and you will see how soon: *GetCapabilities* is used to retrieve service and layer metadata, *GetMap* is used any time the GeoNode user update the layers's view in the map composer (GeoExplorer) or in the layer details page, *GetFeatureInfo* is used any time the GeoNode user identify one or more features from the map composer, *DescribeLayer* is used by the style editor, *GetLegendGraphic* is used to generate the legend elements in the GeoExplorer table of contents, *GetStyles* is used to retrieve the styles associated to a layer

GetCapabilities

To see how the WMS GetCapabilities request is composed, and the result of its output, you can click on this link (which is the same link you find in the GeoServer home page for WMS 1.3.0): http://localhost:8080/geoserver/ows?service=wms&version=1.3.0&request=GetCapabilities

Spend some minutes to observe the XML response of this request: you should recognize:

- an initial section with metadata related to the WMS server: name, title, abstract, keywords, contact information). All of these information may be changed using the GeoServer administrative site
- a section with supported operations: GetCapabilities, GetMap (with the supported output formats), GetFeatureInfo (with the supported output formats)

- a sections with all of the exposed layers and the supported spatial reference system. For each layer the GetCapabilities document returns the name, title, abstract, keywords, bounding box, attribution informations and styles

You should see the four layers which you have uploaded so far to GeoNode, and all of the metadata you compiled.

Now you will use the OWSLib Python library to parse the GetCapabilities document of your GeoNode instance in order to get metadata information about your instance and its layers. Open a new shell, log in the vagrant box, activate the virtualenv and run the Django shell:

```
✗ vagrant ssh
$ . /workshop/env/bin/activate
$ cd /workshop/geonode/
$ python manage.py shell
```

Now using Python and OWSLib open the GeoNode WMS endpoint and check its basic metadata:

```
>>> from owslib.wms import WebMapService
>>> wms = WebMapService('http://localhost:8080/geoserver/ows?') # open GeoNode WMS en
>>> print wms.identification.title # you can change this and the following metadata f
My GeoServer WMS
>>> print wms.identification.abstract
This is a description of your Web Map Server.
>>> print wms.identification.keywords
['WFS', 'WMS', 'GEOSERVER']
>>>
```

Print the WMS supported operation names:

```
>>> for operation in wms.operations:
        print operation.name # print each WMS operation name

GetCapabilities
GetMap
GetFeatureInfo
DescribeLayer
GetLegendGraphic
GetStyles
```

List service's layers:

```
>>> list(wms.contents)
['geonode:biketrails_arc_p',
 'geonode:boston_public_schools_2012_z1l',
 'geonode:socioeconomic_status_2000_2014_9p1',
 'geonode:subwaylines_p_odp']
```

Print main information of a layer:

```
>>> layer = wms['geonode:biketrails_arc_p']
>>> print layer.title
Bike Trails updated
>>> print layer.name
geonode:biketrails_arc_p
>>> print layer.abstract
2009 MBTA bike trails updated
>>> print layer.keywords
['FOSS4G2017', 'commutee']
>>> print layer.boundingBox
(-73.41141983595836, 41.394593205113374, -69.94761156918418, 42.87012688741449, 'EPSG
```

GetMap

The GetMap request can be used to request the WMS endpoint of GeoServer to generate a map. GeoNode use behind the scenes GetMap requests to GeoServer to provide maps to the end users. By default GeoNode, for performance reasons, cache the output of these requests with GeoWebCache, but this beahviour can be changed from the GeoServer administrative interface.

In a WMS GetMap request there are several parameters which are involved, a complete list is here.

The required parameters for a GetMap request are:

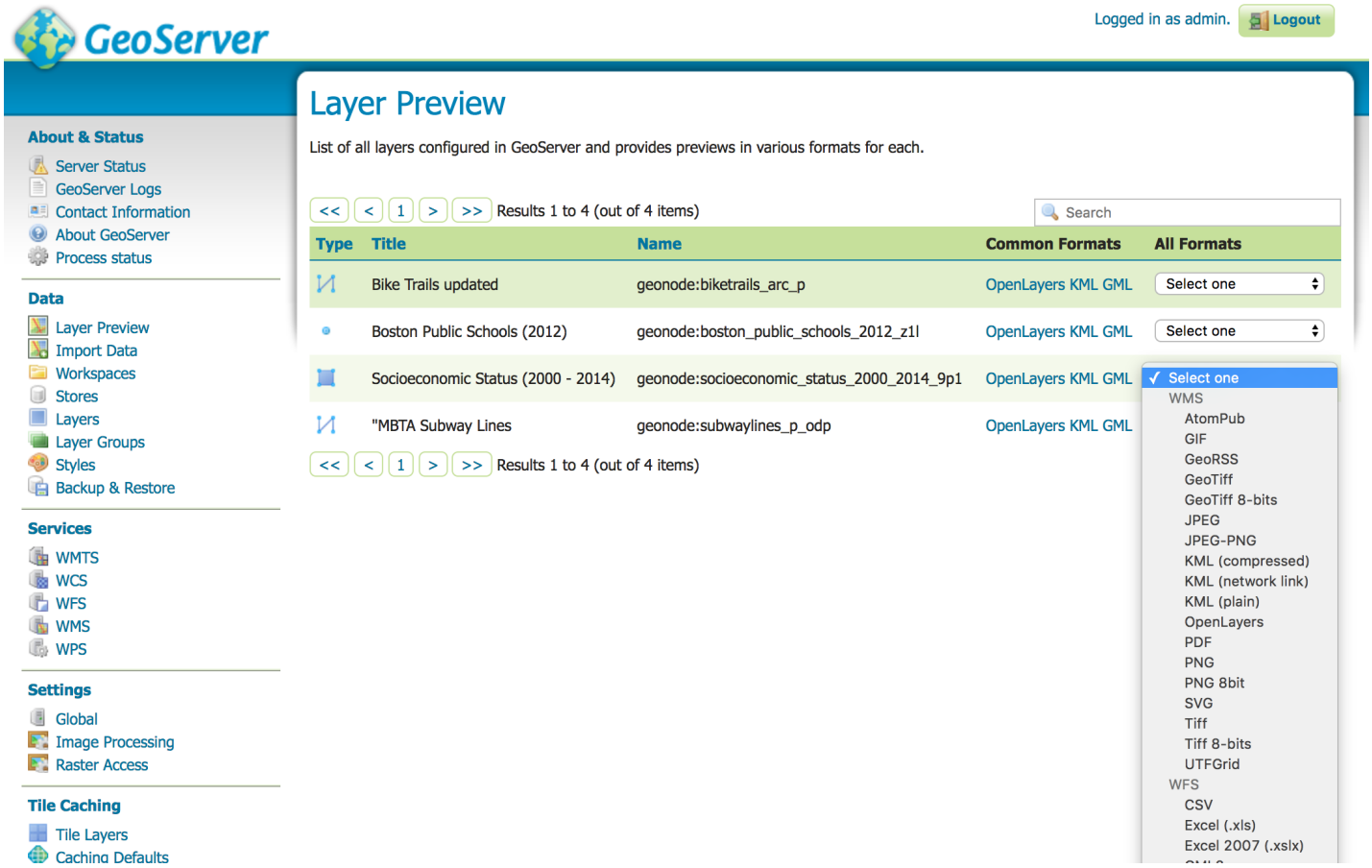
- **service**: always *WMS*
- **version**: Value is one of *1.0.0*, *1.1.0*, *1.1.1*, *1.3.0*
- **request**: always *GetMap*
- **layers**: layers to display on map (comma-separated list of layer names)
- **styles**: styles to use in the map. If empty it uses default styles
- **srs** of **crs**: Spatial Reference System to use in map output, using the EPSG:nnn format
- **bbox** bounding box of the map extent, in the format minx,miny,maxx,maxy
- **width** width of map output in pixels

- **height** height of map output in pixels
- **format** image format for map output

For example here is a GetMap to get a map image of the "Socioeconomic Status (2000-2014)" layer (click on the link to see it): http://localhost:8080/geoserver/wms?LAYERS=geonode%3Asocioeconomic_status_2000_2014_9p1&SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&STYLES=&FORMAT=image%2Fpng&SRS=EPSG%3A900913&BBOX=-7915207,5209947,-7910315,5214839&WIDTH=256&HEIGHT=256

An useful way to build a WMS GetMap request for a layer is by using the GeoServer Layer Preview interface. Go to the GeoServer admin page (http://localhost:8080/geoserver), then click on the *Data > Layer Preview* menu.

Select the layer for which you want to build the WMS GetMap, a format and then the browser will open a new page with the GetMap request in the url



Get back now to the Django shell, and you will use OWSLib to perform the same WMS GetMap request programmatically by using Python:

```
>>> from owslib.wms import WebMapService
>>> wms = WebMapService('http://localhost:8080/geoserver/ows?')
>>> img = wms.getmap(
```

```
...:     layers=['geonode:socioeconomic_status_2000_2014_9p1'],
...:     srs='EPSG:900913',
...:     bbox=(-7915207,5209947,-7910315,5214839),
...:     size=(256, 256),
...:     format='image/png'
...: )
>>> out_img = open('/workshop/test_get_map.png', 'wb')
>>> out_img.write(img.read())
>>> out_img.close()
```

Make sure your image has been correctly generated at */workshop/test_get_map.png*. You can easily open it on your host OS by going to the directory where you cloned the workshop repository.

GetFeatureInfo

The GeoServer WMS GetFeatureInfo request is used by GeoNode to return the spatial and attribute data for the features at a given location on a map. For this purpose an alternative could be to use the WFS GetFeatureInfo request, but the GetMap GetFeatureInfo request provide the advantage that request uses an (x, y) pixel value from a returned WMS image.

In a WMS GetFeatureInfo request there are several parameters which are involved, a complete list is here.

The required parameters for a GetFeatureInfo request are very similar to the ones you have used for the GetMap request. Differences are:

- **request**: always *GetFeatureInfo*
- **query_layers**: a comma-separated list of one or more layers to query
- **x** or **i**: X ordinate of query point on map, in pixels. 0 is left side. i is the parameter key used in WMS 1.3.0
- **y** or **j**: Y ordinate of query point on map, in pixels. 0 is the top. j is the parameter key used in WMS 1.3.0
- **info_format**: Format of the response, can be TEXT, GML2, GML3, HTML, JSON, JSONP. This is optional, default is TEXT

Here is a sample GetFeatureInfo request to the socioeconomic_status_2000_2014_9p1 layer. Click on the link to check the GeoServer response:

http://localhost:8080/geoserver/wms?LAYERS=geonode:socioeconomic_status_2000_2014_9p1&QUERY_LAYERS=geonode:socioeconomic_status_2000_2014_9p1&SERVICE=WMS&VERSION=1.1.1&REQUEST=GetFeatureInfo&B

BOX=-7921478,5202230,-7900993,5214785&HEIGHT=657&WIDTH=1072&FORMAT=image/png&INFO_FORMAT=application/json&SRS=EPSG:900913&X=489&Y=376

DescribeLayer

GeoExplorer, the client in GeoNode, needs to know the structure of the data, for example to implement the style editor widget and the query widget. This is done using a WMS DescribeLayer to GeoServer.

The required parameters are:

- **service**: always *WMS*
- **version**: Value is one of *1.0.0*, *1.1.0*, *1.1.1*, *1.3.0*
- **request**: always *DescribeLayer*
- **layers**: layers to query (comma-separated list of layer names)

Here is a sample DescribeLayer request to the socioeconomic_status_2000_2014_9p1 layer. Click on the link to check the GeoServer response:

http://localhost:8080/geoserver/wms?LAYERS=geonode%3Asocioeconomic_status_2000_2014_9p1&SERVICE=WMS&VERSION=1.1.1&REQUEST=DescribeLayer&outputFormat=text/xml

The output will return the GeoServer service type (WFS for vector layers, WCS for coverage) endpoint to use for getting more information about the layer.

In this case it is a WFS. Later in this tutorial you will have a look at how to interact with WFS in GeoServer.

GetLegendGraphic

The WMS GetLegendGraphic operation is used by GeoNode to generate the legends in the map composer and in the layer page.

Here is the GetLegendGraphic request for the “Socioeconomic Status (200-2014)” layer:

http://localhost:8080/geoserver/wms?TRANSPARENT=TRUE&SERVICE=WMS&VERSION=1.1.1&REQUEST=GetLegendGraphic&TILED=true&LAYER=geonode:socioeconomic_status_2000_2014_9p1&transparent=true&format=image/png

GetLegendGraphic supports a large number of parameters and options to let a client to generate a great legend graphic. For a full list you can have a read here

GetStyles

The WMS GetStyles operation is used by GeoNode to retrieve the list of styles associated to a layer.

Here is the GetStyles request for the “Socioeconomic Status (200-2014)” layer:

http://localhost:8080/geoserver/wms?SERVICE=WMS&VERSION=1.1.1&REQUEST=GetStyles&LAYERS=geonode:socioeconomic_status_2000_2014_9p1

WFS

> The Web Feature Service (WFS) is a standard created by the Open Geospatial Consortium (OGC) for creating, modifying and exchanging vector format geographic information on the Internet using HTTP. A WFS encodes and transfers information in Geography Markup Language (GML), a subset of XML. The current version of WFS is 2.0.0. GeoServer supports versions 2.0.0, 1.1.0, and 1.0.0. Although there are some important differences between the versions, the request syntax often remains the same.

> The WFS standard defines the framework for providing access to, and supporting transactions on, discrete geographic features in a manner that is independent of the underlying data source. Through a combination of discovery, query, locking, and transaction operations, users have access to the source spatial and attribute data in a manner that allows them to interrogate, style, edit (create, update, and delete), and download individual features. The transactional capabilities of WFS also support the development and deployment of collaborative mapping applications.

For an extended reference to the WFS standard in GeoServer check the GeoServer WFS reference documentation

GeoNode uses the version 1.1.0 of the WFS standard, for which GeoServer implements the following operations (the last three ones are optional for this standard, but implemented in GeoServer):

- **GetCapabilities** retrieves metadata about the service, including supported operations and parameters, and a list of the available vector layers
- **DescribeFeatureType** returns a description of feature types supported by a WFS service
- **GetFeature** returns a selection of features from a data source including geometry and attribute values
- **LockFeature** prevents a feature from being edited through a persistent feature lock
- **Transaction** edits existing feature types by creating, updating, and deleting

GeoNode implements all of these operations but not the LockFeature: *GetCapabilities* is used to retrieve service and layers metadata, *DescribeFeatureType* is used by the style editor to read the information of the layer attributes (field name, type...), *GetFeature* is used when downloading the dataset,by the query builder and by the identify tool, *Transaction* is used when interacting with the editing tools.

GetCapabilities

You can see the WFS GetCapabilities response using this link:

http://localhost:8080/geoserver/ows?service=wfs&version=1.1.0&request=GetCapabilities

Spend some time inspecting the content of the output of this request. You should recognize:

- an initial session with ServiceIdentification: this contains the metadata of the WFS service
- a session with ServiceProvider: this provides metadata related to the contact point
- a session with supported operations
- finally a session containing all of the GeoServer vector layers you uploaded. For each layer are provided metadata such as the layer's name, title, abstract, keywords, default SRS and the bounding box

As you did with WMS, try to use OWSLib with Python to get to these metadata:

```
>>> from owslib.wfs import WebFeatureService
>>> wfs = WebFeatureService(url='http://localhost:8080/geoserver/ows?', version='1.1.
>>> print wfs.identification.title
My GeoServer WFS
>>> print wfs.identification.abstract
This is a description of your Web Feature Server.
...
>>> print wfs.identification.keywords
['WFS', 'WMS', 'GEOSERVER']
```

Print the WFS supported operation names:

```
>>> for operation in wfs.operations:
      print operation.name
GetCapabilities
DescribeFeatureType
GetFeature
GetGmlObject
LockFeature
GetFeatureWithLock
Transaction
```

List the service's layers:

```
>>> list(wfs.contents)
['geonode:boston_public_schools_2012_z1l',
 'geonode:biketrails_arc_p',
 'geonode:socioeconomic_status_2000_2014_9p1',
 'geonode:subwaylines_p_odp']
```

Get the metadata of a layer:

```
>>> layer = wfs['geonode:subwaylines_p_odp']
>>> print layer.id
geonode:subwaylines_p_odp
>>> print layer.title
"MBTA Subway Lines
>>> print layer.abstract
MBTA Subway Lines (Massachusetts Bay Transportation Authority)
>>> print layer.keywords
['FOSS4G2017', 'boston', 'transportation']
>>> print layer.boundingBoxWGS84
(-71.25301493328648, 42.207496931441305, -70.99102173442837, 42.437486977030176)
```

DescribeFeatureType

DescribeFeatureType returns a description of feature types supported by a WFS service

http://localhost:8080/geoserver/wfs?SERVICE=WFS&REQUEST=DescribeFeatureType&TYPENAME=geonode%3Asocioeconomic_status_2000_2014_9p1

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:geonode="http://www.ge
  <xsd:import namespace="http://www.opengis.net/gml/3.2" schemaLocation="http://local
  <xsd:complexType name="socioeconomic_status_2000_2014_9p1Type">
    <xsd:complexContent>
      <xsd:extension base="gml:AbstractFeatureType">
        <xsd:sequence>
          <xsd:element maxOccurs="1" minOccurs="0" name="the_geom" nillable="true" ty
          <xsd:element maxOccurs="1" minOccurs="0" name="CT_ID_1" nillable="true" typ
          <xsd:element maxOccurs="1" minOccurs="0" name="GEOID10" nillable="true" typ
          <xsd:element maxOccurs="1" minOccurs="0" name="ALAND10" nillable="true" typ
          <xsd:element maxOccurs="1" minOccurs="0" name="AWATER1" nillable="true" typ
```

```
            <xsd:element maxOccurs="1" minOccurs="0" name="POP100" nillable="true" type
            <xsd:element maxOccurs="1" minOccurs="0" name="HU100" nillable="true" type=
            <xsd:element maxOccurs="1" minOccurs="0" name="Type" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="Res" nillable="true" type="x
            <xsd:element maxOccurs="1" minOccurs="0" name="BRA_PD_" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="BRA_PD" nillable="true" type
            <xsd:element maxOccurs="1" minOccurs="0" name="Cty_Cnc" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="WARD" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="ISD_Nbh" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="Polc_Ds" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="Fr_Dstr" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="PWD" nillable="true" type="x
            <xsd:element maxOccurs="1" minOccurs="0" name="HI00" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="MR00" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="HI10" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="MR10" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="HI00_10" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="MR00_10" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="HI14" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="MR14" nillable="true" type="
            <xsd:element maxOccurs="1" minOccurs="0" name="HI10_14" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="MR10_14" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SESdx00" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SES00tr" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SESdx10" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SES10tr" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SESdx14" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SES14tr" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SES00_1" nillable="true" typ
            <xsd:element maxOccurs="1" minOccurs="0" name="SES10_1" nillable="true" typ
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="socioeconomic_status_2000_2014_9p1" substitutionGroup="gml:Abstr
  </xsd:schema>
```

GetFeature

The request parameters for GetFeature are sent with a POST HTTP Request to the WFS endpoint (typically http://localhost:8080/geoserver/wfs?).

Request:

```
  <wfs:GetFeature xmlns:wfs="http://www.opengis.net/wfs" service="WFS" version="1.1.0"
    <wfs:Query typeName="feature:collegesuniversities_gap" srsName="EPSG:900913" xmlns:
      <ogc:Filter xmlns:ogc="http://www.opengis.net/ogc"><ogc:FeatureId fid="collegesun
```

```
    </wfs:Query>
  </wfs:GetFeature>
```

Response:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <wfs:FeatureCollection xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:wfs="http://
  <gml:featureMembers>
    <geonode:collegesuniversities_gap gml:id="collegesuniversities_gap.47">
      <geonode:the_geom>
        <gml:Point srsName="http://www.opengis.net/gml/srs/epsg.xml#900913" srsDimensio
        <gml:pos>-7902163.8380227 5209832.6273378</gml:pos>
      </gml:Point>
    </geonode:the_geom>
    <geonode:COLLEGE>College XYZ</geonode:COLLEGE>
    <geonode:City>Boston</geonode:City>
    <geonode:State>MA</geonode:State>
  </geonode:collegesuniversities_gap>
  </gml:featureMembers>
  </wfs:FeatureCollection>
```

You can use OWSLib to run GetFeature from Python:

```
  >>> response = wfs.getfeature(typename='geonode:boston_public_schools_2012_z1l')
  >>> response.read()
```

Transaction

GeoNode use the *Transaction* operation when editing layers. The request parameters are sent with a POST HTTP Request to the WFS endpoint (typically http://localhost:8080/geoserver/wfs?).

A *Transaction* can be an **Insert**, **Update** or **Delete**. More features can be inserted, updated or deleted within the same transaction.

wfs:Insert

a *wfs:Insert* transaction is used to add one or more features to a given layer

Request:

```
  <wfs:Transaction xmlns:wfs="http://www.opengis.net/wfs" service="WFS" version="1.1.0"
    <wfs:Insert>
```

```
      <feature:collegesuniversities_gap xmlns:feature="http://www.geonode.org/">
        <feature:the_geom>
          <gml:Point xmlns:gml="http://www.opengis.net/gml" srsName="EPSG:900913"><gml:
        </feature:the_geom>
        <feature:COLLEGE>College XYZ</feature:COLLEGE>
        <feature:City>Boston</feature:City>
        <feature:State>MA</feature:State>
      </feature:collegesuniversities_gap>
    </wfs:Insert>
  </wfs:Transaction>
```

Response:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <wfs:TransactionResponse xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:wfs="http:
    <wfs:TransactionSummary>
      <wfs:totalInserted>1</wfs:totalInserted>
      <wfs:totalUpdated>0</wfs:totalUpdated>
      <wfs:totalDeleted>0</wfs:totalDeleted>
    </wfs:TransactionSummary>
    <wfs:TransactionResults/>
    <wfs:InsertResults>
      <wfs:Feature><ogc:FeatureId fid="collegesuniversities_gap.47"/></wfs:Feature>
  </wfs:InsertResults>
  </wfs:TransactionResponse>
```

wfs:Update

a *wfs:Update* transaction is used to update one or more features in a given layer

Request:

```
  <wfs:Transaction xmlns:wfs="http://www.opengis.net/wfs" service="WFS" version="1.1.0"
    <wfs:Update typeName="feature:collegesuniversities_gap" xmlns:feature="http://www.g
      <wfs:Property>
        <wfs:Name>COLLEGE</wfs:Name>
        <wfs:Value>College 123</wfs:Value>
      </wfs:Property>
      <ogc:Filter xmlns:ogc="http://www.opengis.net/ogc">
        <ogc:FeatureId fid="collegesuniversities_gap.47"/>
      </ogc:Filter>
    </wfs:Update>
  </wfs:Transaction>
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<wfs:TransactionResponse xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:wfs="http:
  <wfs:TransactionSummary>
    <wfs:totalInserted>0</wfs:totalInserted>
    <wfs:totalUpdated>1</wfs:totalUpdated>
    <wfs:totalDeleted>0</wfs:totalDeleted>
  </wfs:TransactionSummary>
  <wfs:TransactionResults/>
  <wfs:InsertResults>
    <wfs:Feature>
      <ogc:FeatureId fid="none"/>
    </wfs:Feature>
  </wfs:InsertResults>
</wfs:TransactionResponse>
```

wfs:Delete

a *wfs:Delete* transaction is used to remove one or more features from a given layer

Request:

```
<wfs:Transaction xmlns:wfs="http://www.opengis.net/wfs" service="WFS" version="1.1.0"
  <wfs:Delete typeName="feature:collegesuniversities_gap" xmlns:feature="http://www.g
    <ogc:Filter xmlns:ogc="http://www.opengis.net/ogc">
      <ogc:FeatureId fid="collegesuniversities_gap.47"/>
    </ogc:Filter>
  </wfs:Delete>
</wfs:Transaction>
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<wfs:TransactionResponse xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:wfs="http:
  <wfs:TransactionSummary>
    <wfs:totalInserted>0</wfs:totalInserted>
    <wfs:totalUpdated>0</wfs:totalUpdated>
    <wfs:totalDeleted>1</wfs:totalDeleted>
  </wfs:TransactionSummary>
  <wfs:TransactionResults/>
  <wfs:InsertResults>
    <wfs:Feature>
      <ogc:FeatureId fid="none"/>
    </wfs:Feature>
```

```
    </wfs:InsertResults>
</wfs:TransactionResponse>
```

The GeoServer REST API

GeoExplorer, the default GeoNode mapping client, uses OGC standards whenever is possible to submit requests to GeoServer. In fact GeoExplorer was initially developed to interact with GeoServer by itself. GeoNode is adding more features on top of it, mainly CMS abilities and granular permissions implementation.

But there are things that cannot be done using OGC standards: for example uploading a shapefile or a TIFF dataset as a new layer, or modifying styles for a layer. For doing this, GeoNode uses the GeoServer REST API.

> GeoServer provides a RESTful interface through which clients can retrieve information about an instance and make configuration changes. Using the REST interface's simple HTTP calls, clients can configure GeoServer without needing to use the Web administration interface.

> REST is an acronym for "REpresentational State Transfer". REST adopts a fixed set of operations on named resources, where the representation of each resource is the same for retrieving and setting information. In other words, you can retrieve (read) data in an XML format and also send data back to the server in similar XML format in order to set (write) changes to the system.

> Operations on resources are implemented with the standard primitives of HTTP: GET to read; and PUT, POST, and DELETE to write changes. Each resource is represented as a URL, such as http://GEOSERVER_HOME/rest/workspaces/topp.

The URL at which you can find the GeoServer API is http://localhost:8080/geoserver/rest

Using this URL you can read and make changes to the GeoServer configuration with GET, PUT, POST and DELETE requests.

Here are some samples GET request, useful to retrieve configuration information (make sure to be authenticated in the GeoServer administrative site):

- list of layers: http://localhost:8080/geoserver/rest/layers
- configuration for a layer: http://localhost:8080/geoserver/rest/layers/boston_public_schools_2012_z1l.html
- list of styles: http://localhost:8080/geoserver/rest/styles

- configuration for a style: http://localhost:8080/geoserver/rest/styles/biketrails_arc_p.html

Playing with the GeoServer REST API using curl

To see some samples with PUT, POST and DELETE you are going to use curl, a very useful command line and library for transferring data with URLs. In case you want more information about curl you can check out the Everything curl guide

You will now use curl to perform PUT, POST and DELETE requests to the GeoServer REST API. You will use a POST request to create a new GeoServer style (you can find the style source file, in the SLD format, in */workshop/data/sld/my_style.sld*), a PUT request to modify an existing GeoServer style and a DELETE request to delete a GeoServer style.

As a first thing, you will create the new style in GeoServer with a POST request:

```
$ curl -v -u admin:geoserver -XPOST -H "Content-type: text/xml" -d "<style><name>my_s
```

If now you go to http://localhost:8080/geoserver/rest/styles you should see the link to the new style (*my_style*) there. Now you will upload the my_style.sld file to GeoServer using a PUT request:

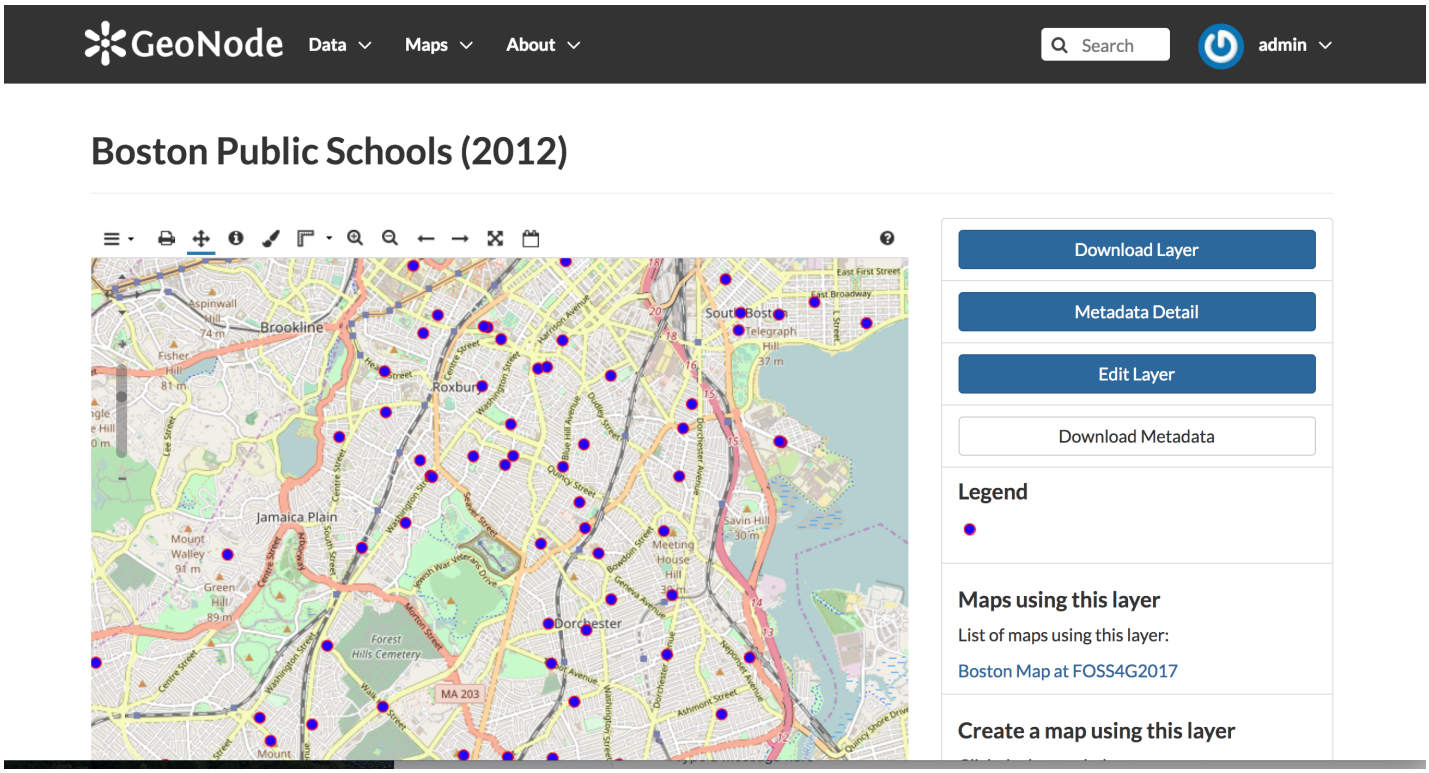
```
$ curl -v -u admin:geoserver -XPUT -H "Content-type: application/vnd.ogc.sld+xml" -d
```

If you now go to http://localhost:8080/geoserver/rest/styles/my_style.html you should see a link to the *my_style.sld* file.

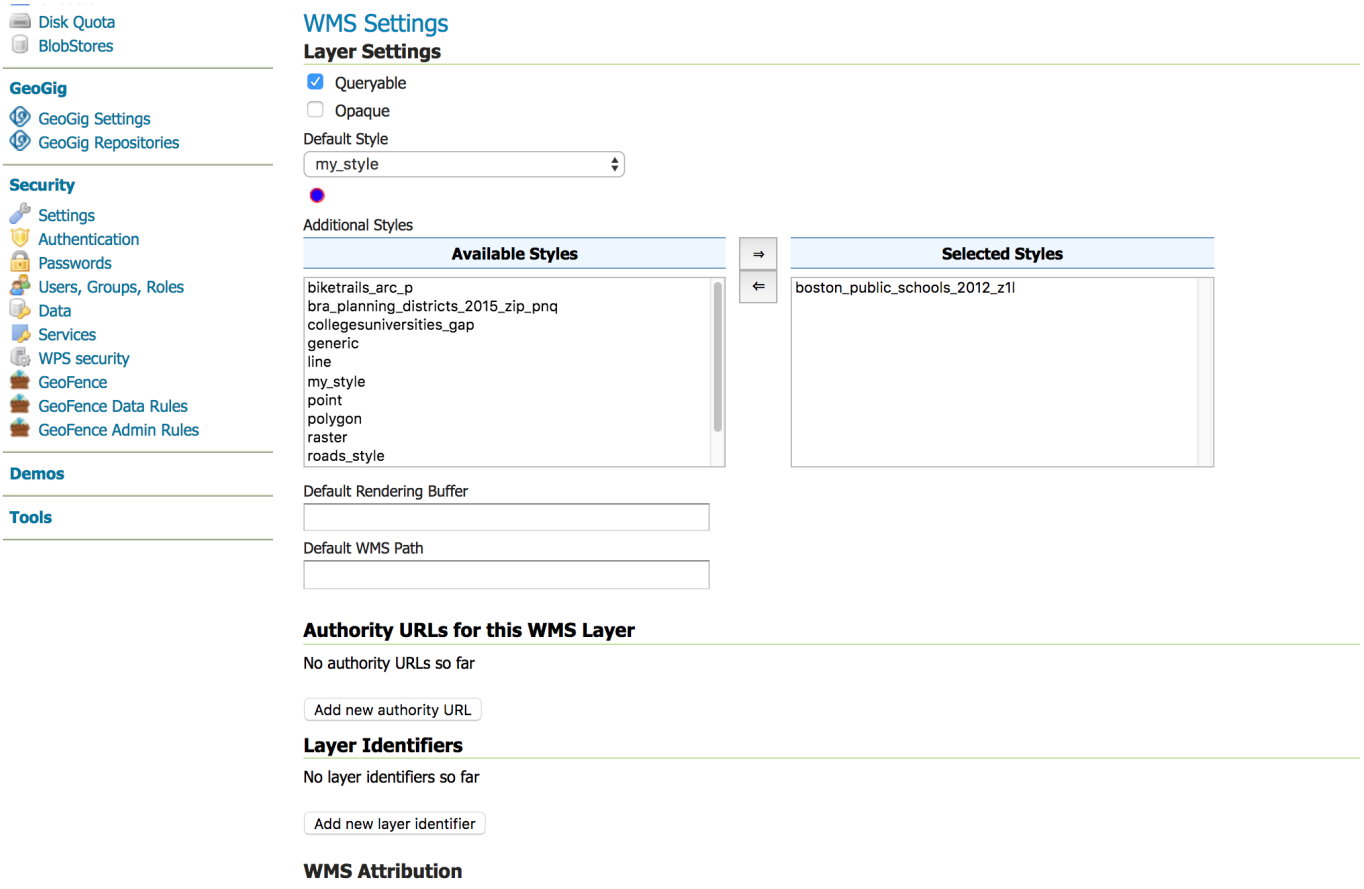
Now you can associate the style to the *boston_public_schools_2012_z1l* layer with a PUT request:

```
curl -u admin:geoserver -XPUT -H 'Content-type: text/xml' -d '<layer><defaultStyle><n
```

If you go to the GeoNode layer page you should see *my_style.sld* as the new default style for the *boston_public_schools_2012_z1l* layer



Another way to check if the default style has been changed for the layer is by looking at the GeoSever layer page, in the *Publishing* section



And, of course, you can see which default style is being used by using the REST API: http://localhost:8080/geoserver/rest/layers/boston_public_schools_2012_z1l.html

Now you are going to set the default style back to the previous one, *boston_public_schools_2012_z1l*:

```
curl -u admin:geoserver -XPUT -H 'Content-type: text/xml' -d '<layer><defaultStyle><n
```

Finally use a DELETE request to remove the *my_style* style:

```
curl -u admin:geoserver -XDELETE http://localhost:8080/geoserver/rest/styles/my_style
```

Check if the style was removed using the GeoServer administrative site or the GeoServer REST API.

Playing with the GeoServer REST API using gsconfig

Similarly to what you did with styles, you can use the GeoServer REST API to read and modify configuration for GeoServer workspaces, stores, layers and many other things. As said GeoNode uses the REST API for a series of things which happens behind the scenes. GeoNode interact with the REST API using the gsconfig Python library, developed by Boundless.

Using the Django shell, you will now use the gsconfig which comes with GeoNode to query the GeoServer catalog, and to modify its configuration using the Python language.

Import gsconfig and connect to the GeoServer REST API:

```
>>> from geoserver.catalog import Catalog
>>> cat = Catalog('http://localhost:8080/geoserver/rest', 'admin', 'geoserver')
```

Get the store names:

```
>>> for store in cat.get_stores():
>>>     print store.name
biketrails_arc_p
boston_public_schools_2012_z1l
socioeconomic_status_2000_2014_9p1
subwaylines_p_odp
```

Get information about a store:

```
store = cat.get_store('biketrails_arc_p')
>>> print store.resource_type
dataStore
>>> print store.type
Shapefile
>>> print store.workspace
geonode @ http://localhost:8080/geoserver/rest/workspaces/geonode.xml
>>> for resource in store.get_resources(): # a shapefile store as just one resource,
    print resource.name
biketrails_arc_p
```

Get information about a layer:

```
>>> layer = cat.get_resource('biketrails_arc_p')
>>> print layer.name
biketrails_arc_p
>>> print layer.title
Bike Trails updated
>>> print layer.abstract
2009 MBTA bike trails updated
>>> print layer.advertised
true
>>> print layer.keywords
['FOSS4G2017', 'commutee']
>>> print layer.latlon_bbox
('-73.41141983595836', '-69.94761156918418', '41.394593205113374', '42.87012688741449
>>> print layer.projection
EPSG:4326
>>> print layer.resource_type
featureType
>>> print layer.attributes
['the_geom', 'TRAIL_STAT', 'OWNER', 'PREV_OWNER', 'MANAGER', 'STATUS_OWN', 'STATUS_MA
```

Get information about styles:

```
>>> for style in cat.get_styles(): # iterate all styles in catalogue
>>>     print style.name
biketrails_arc_p
boston_public_schools_2012_z1l
generic
line
point
polygon
raster
socioeconomic_status_2000_2014_9p1
```

```
subwaylines_p_odp
roads_style

>>> style = cat.get_style('socioeconomic_status_2000_2014_9p1') # access a single sty
>>> print style.name
socioeconomic_status_2000_2014_9p1
>>> print style.filename
socioeconomic_status_2000_2014_9p1.sld

>>> print style.sld_body
<?xml version="1.0" encoding="UTF-8"?>
<sld:StyledLayerDescriptor xmlns="http://www.opengis.net/sld" xmlns:sld="http://www.o
  <sld:NamedLayer>
    <sld:Name>socioeconomic_status_2000_2014_9p1</sld:Name>
    <sld:UserStyle>
      <sld:Name>socioeconomic_status_2000_2014_9p1</sld:Name>
      <sld:Title>socioeconomic_status_2000_2014_9p1</sld:Title>
      <sld:FeatureTypeStyle>
        <sld:Name>name</sld:Name>
        <sld:Rule>
          <sld:PolygonSymbolizer>
            <sld:Fill>
              <sld:CssParameter name="fill">#000088</sld:CssParameter>
            </sld:Fill>
            <sld:Stroke>
              <sld:CssParameter name="stroke">#bbbbff</sld:CssParameter>
              <sld:CssParameter name="stroke-width">0.7</sld:CssParameter>
            </sld:Stroke>
          </sld:PolygonSymbolizer>
        </sld:Rule>
      </sld:FeatureTypeStyle>
    </sld:UserStyle>
  </sld:NamedLayer>
</sld:StyledLayerDescriptor>
```

It is possible to use gsconfig also to modify configurations. GeoNode for example uses gsconfig to modify styles when using the style editor in GeoExplore. Another use of gsconfig in GeoNode is when uploading a new layer to GeoServer: gsconfig is used to create the new store (if needed) and the new layer, and to synchronize the metadata from the Django database to the GeoServer data directory.

Here you will modify the title for one of the layer:

```
>>> print layer.title
'Bike Trails updated'
>>> layer.title = 'Bike Trails updated from gsconfig'
>>> cat.save(layer)
```

You can check if this was effective using GeoServer administrative site, using the REST API or by using gsconfig again and re-reading the layer.

Note: if you want the title synced in GeoNode you will need to use the *updatelayers* GeoNode administrative command, as you will see in a following tutorial.

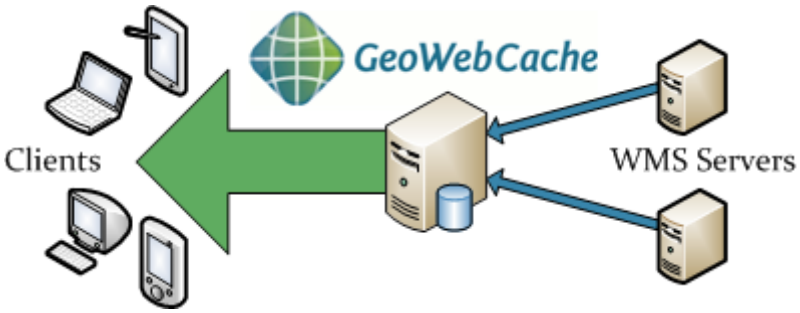
foss4g_2017_geonode_solr

Tiles caching with GeoWebCache

WSM GetMap requests may be time and resource consuming. As many times underlying data are not changing and tiles generated by WMS are always the same, it makes sense to cache them. That is where GeoWebCache comes handy.

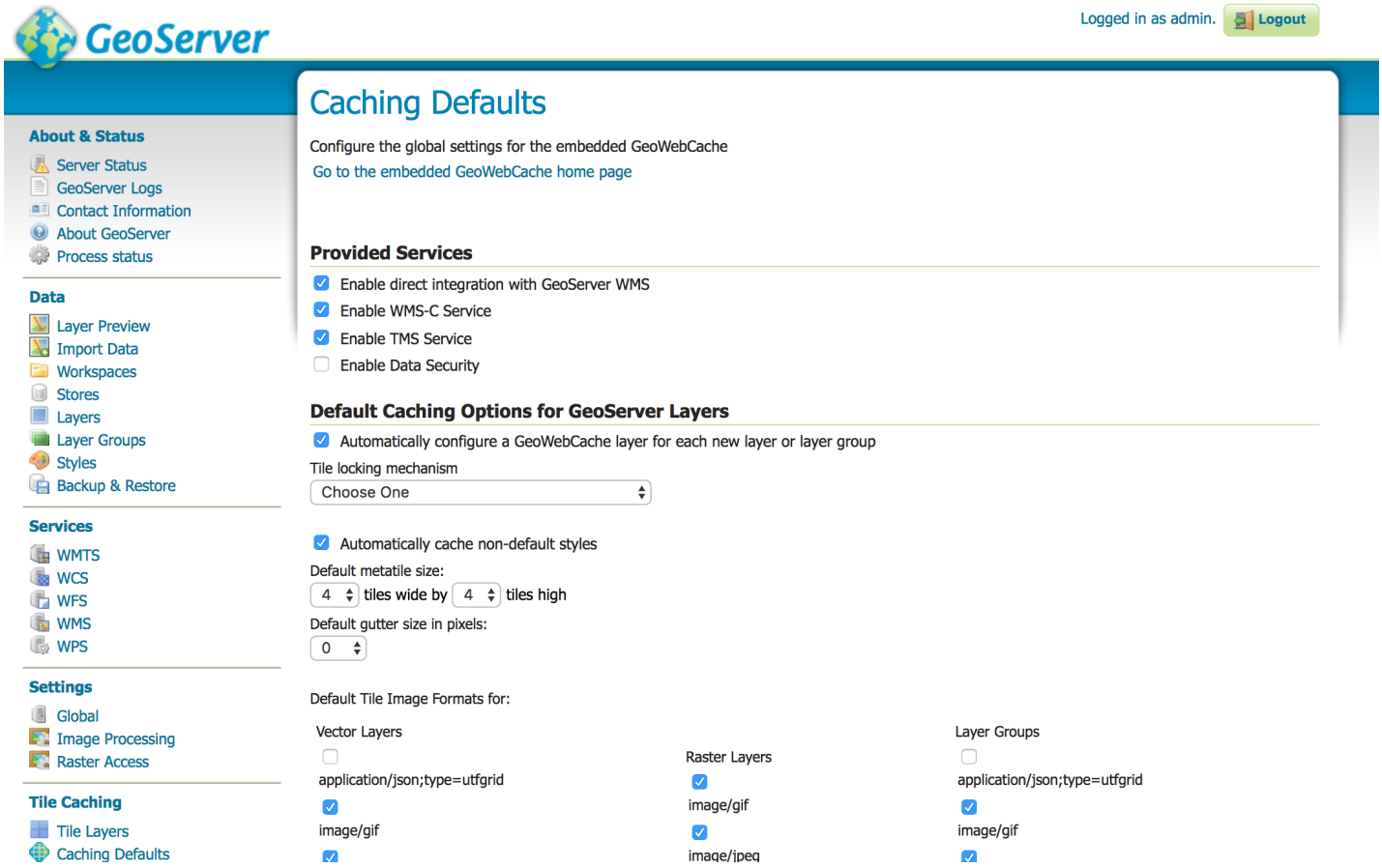
> GeoWebCache is a Java web application used to cache map tiles coming from a variety of sources such as OGC Web Map Server (WMS). It implements various service interfaces (such as WMS-C, WMTS, TMS, Google Maps KML, Virtual Earth) in order to accelerate and optimize map image delivery. It can also recombine tiles to work with regular WMS clients.

> Maps are often static. As most mapping clients render WMS (Web Map Service) data every time they are queried, this can result in unnecessary processing and increased wait times. GeoWebCache optimizes this experience by saving (caching) map images, or tiles, as they are requested, in effect acting as a proxy between client (such as OpenLayers or Google Maps) and server (such as GeoServer, or any WMS-compliant server). As new maps and tiles are requested, GeoWebCache intercepts these calls and returns pre-rendered tiles if stored, or calls the server to render new tiles as necessary. Thus, once tiles are stored, the speed of map rendering increases by many times, creating a much improved user experience.



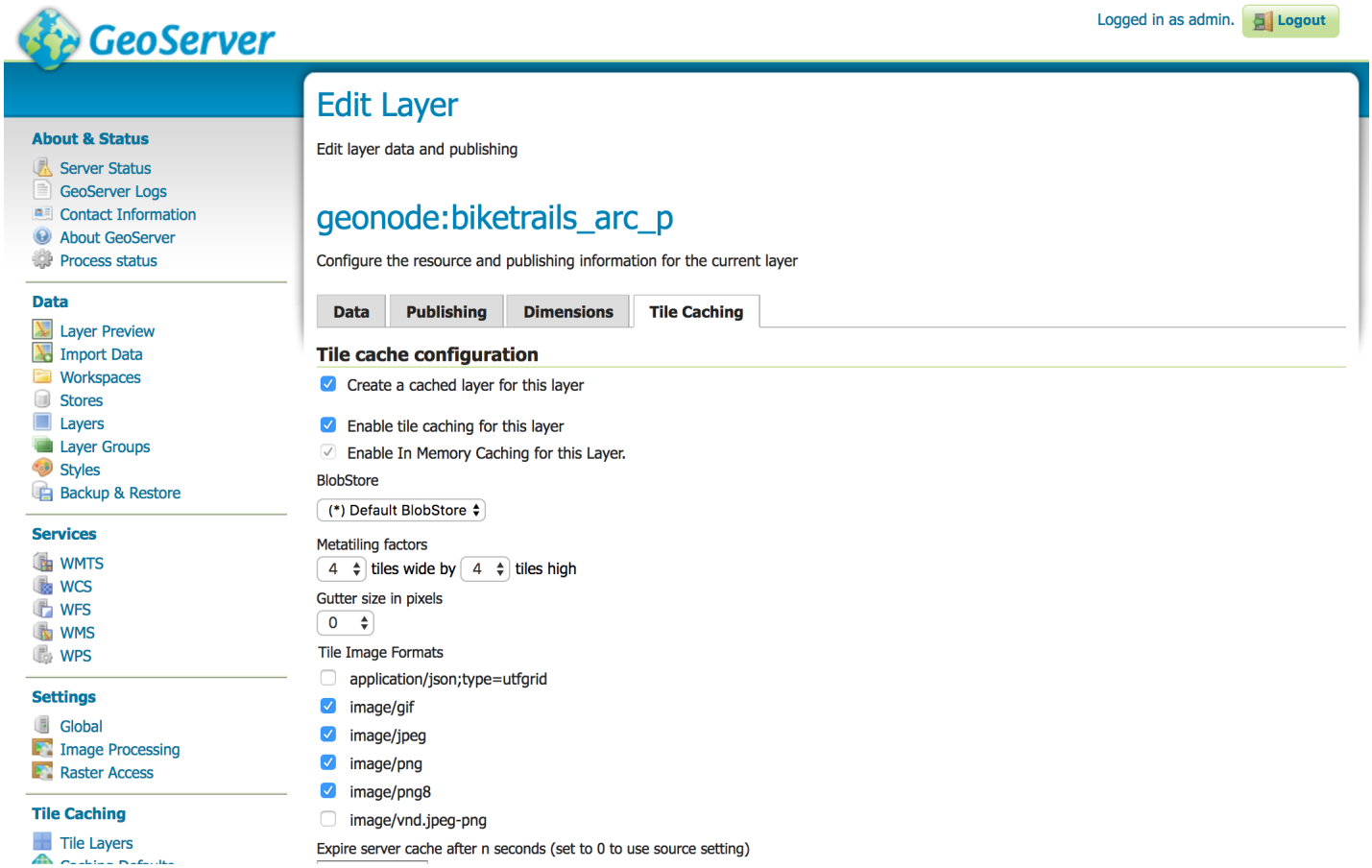
For more information on GeoWebCache (GWC) you can have a look at the official documentation or at the Using GeoWebCache from GeoServer official documentation

GWC is very well integrated in GeoServer (and thus in GeoNode). On GeoNode, by default, WMS requests are cascaded to the GWC endpoint. This behavior can be altered using the *Caching Defaults* session of the GeoServer administrative site.



As you can see, by default the direct integration of GWC with GeoServer WMS is enabled, and each layer which is uploaded to GeoNode is automatically configured for being cached in GWC.

If you look at the GeoServer layer page for one of the layers, you will see that in the *Tile Caching* section the options for creating a cached layer and for enabling tile caching for that layer are enabled by default.



GeoServer WMS will cache and retrieve tiles from GeoWebCache using a GetMap request only if all of these criteria are followed:

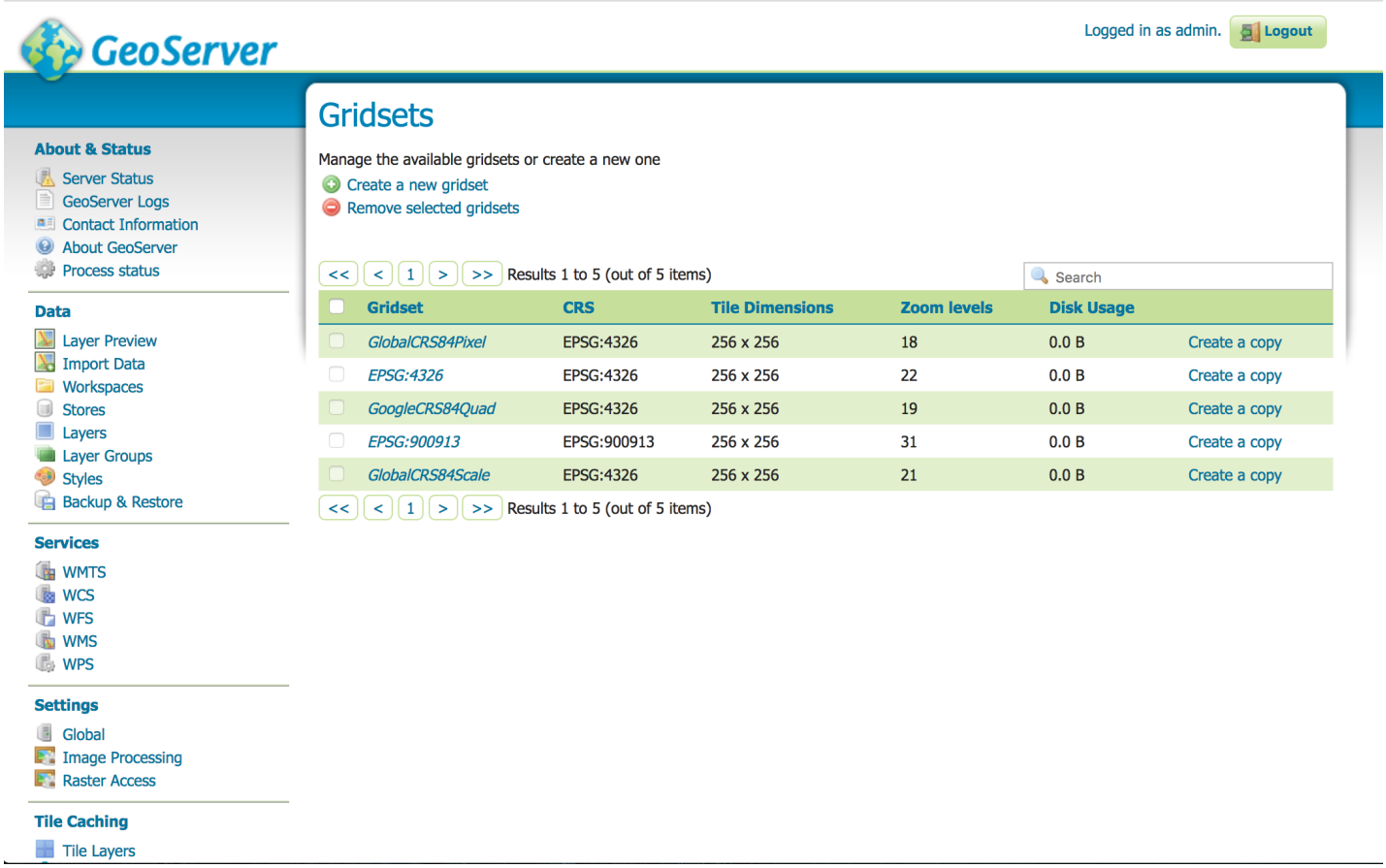
- WMS Direct integration is enabled (which is the case for standard GeoNode, as previously described)
- caching is enabled for that layer (which is the case for standard GeoNode, as previously described)
- request only references a single layer
- tiled=true is included in the request
- image requested is of the same height and width as the size saved in the layer configuration
- requested CRS matches one of the available tile layer gridsets
- image requested lines up with the existing grid bounds
- a parameter is included for which there is a corresponding Parameter Filter

All of these criteria are respected by GeoNode: for example look at this WMS request which is run from the mapping client in the layer page for the “Socioeconomic Status (2000 - 2014) layer (http://localhost:8000/layers/geonode:socioeconomic_status_2000_2014_9p1):

http://localhost:8080/geoserver/wms?LAYERS=geonode%3Asocioeconomic_status_2000_2014_9p1&TRANSPARENT=TRUE&SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&STYLES=&FORMAT=image%2Fpng&TILED=true&SRS=EPSG%3A900913&BBOX=-7912761.16698,5209947.8471924,-7910315.1820752,5212393.8320972&WIDTH=256&HEIGHT=256

Let's analyze if the criteria are respected (the first two ones are already):

- the request reference just a layer: *geonode:socioeconomic_status_2000_2014_9p1*
- TILED=true is in the request
- width (256) and height (256) correspond to available gridsets, as it can be checked in the GeoServer administrative site: *Tile Caching* > *Gridsets*



- requested CRS (EPSG:900913) is one of the available tile layer gridsets, as it can be checked in the GeoServer administrative site: *Layers* > *Layer Name* > *Tile Caching*

- image requested lines up with the existing grid bounds, as it is OpenLayers which is taking care of this
- there are not parameter filters in the request

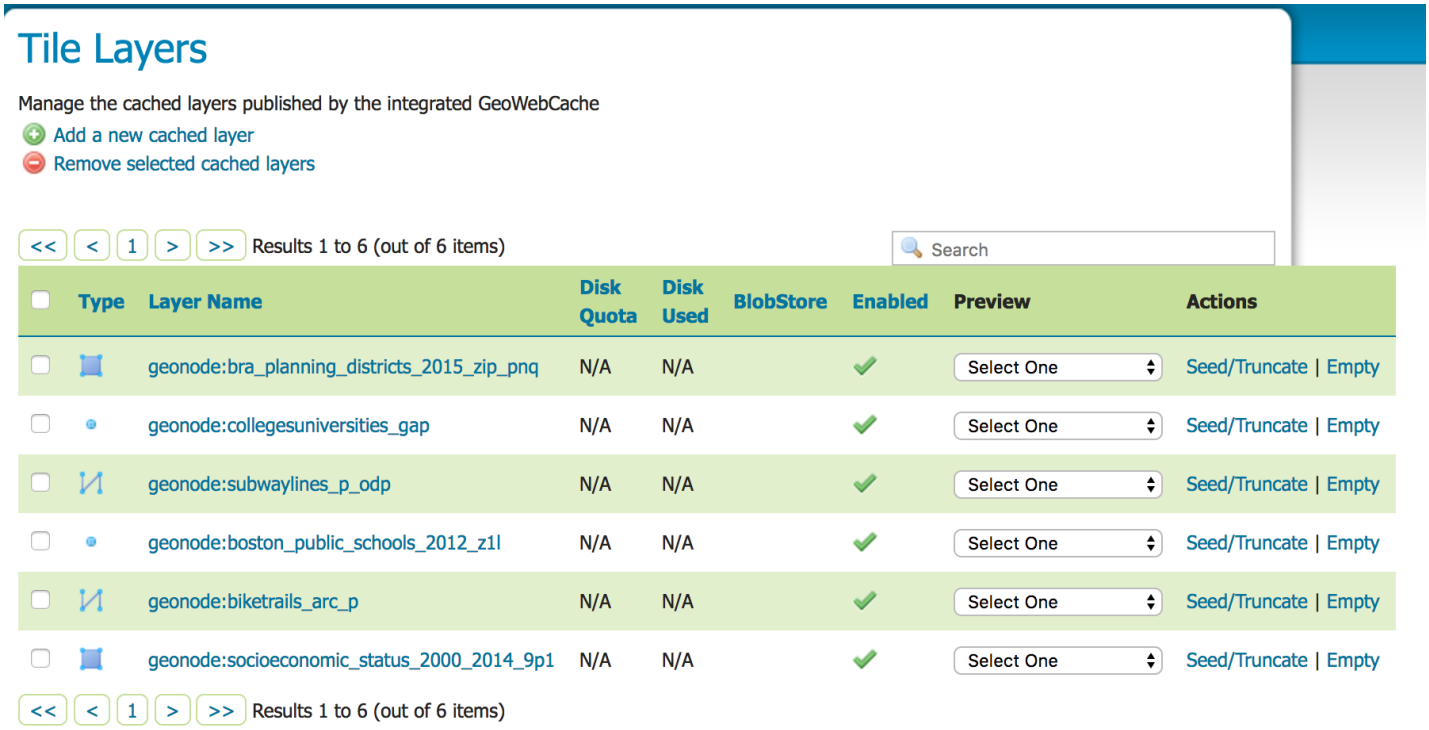
GeoWebCache GeoServer interface

GWC is integated directly in GeoServer and in the GeoServer administrative site. Now you will have a quick tour of the GeoServer interface pages that interacts with GWC. All of these pages are linked in the *Tile Caching* section of the left menu of the interface:

Tile Layers

The *Tile Layers* page shows a list of the layers which are integrated in GWC, and some of their properties:

- *Disk quota* provides the maximum amount of disk space which can be used for a given layer. It will be N/A by default, until *Disk Quotas* are enabled
- *Disk used* is the disk space being used by the tiles generated for a given layer
- *Enabled* indicates if a given layer has tile caching enabled
- *Preview* provides, similarly to the *Layer Preview* for WMS, a simple OpenLayers application to browse the tiles of a layer for each given gridsets (combination of spatial reference and output format)
- *Seed/Truncate* opens the GWC for automatically seeding and truncating the tile cache. This is useful if you need to pre-populate the cache for a layer
- *Empty* removes all of the cached tiles for a given layer



Tile Layers

Manage the cached layers published by the integrated GeoWebCache

Add a new cached layer

Remove selected cached layers

<< < 1 > >> Results 1 to 6 (out of 6 items)

Search

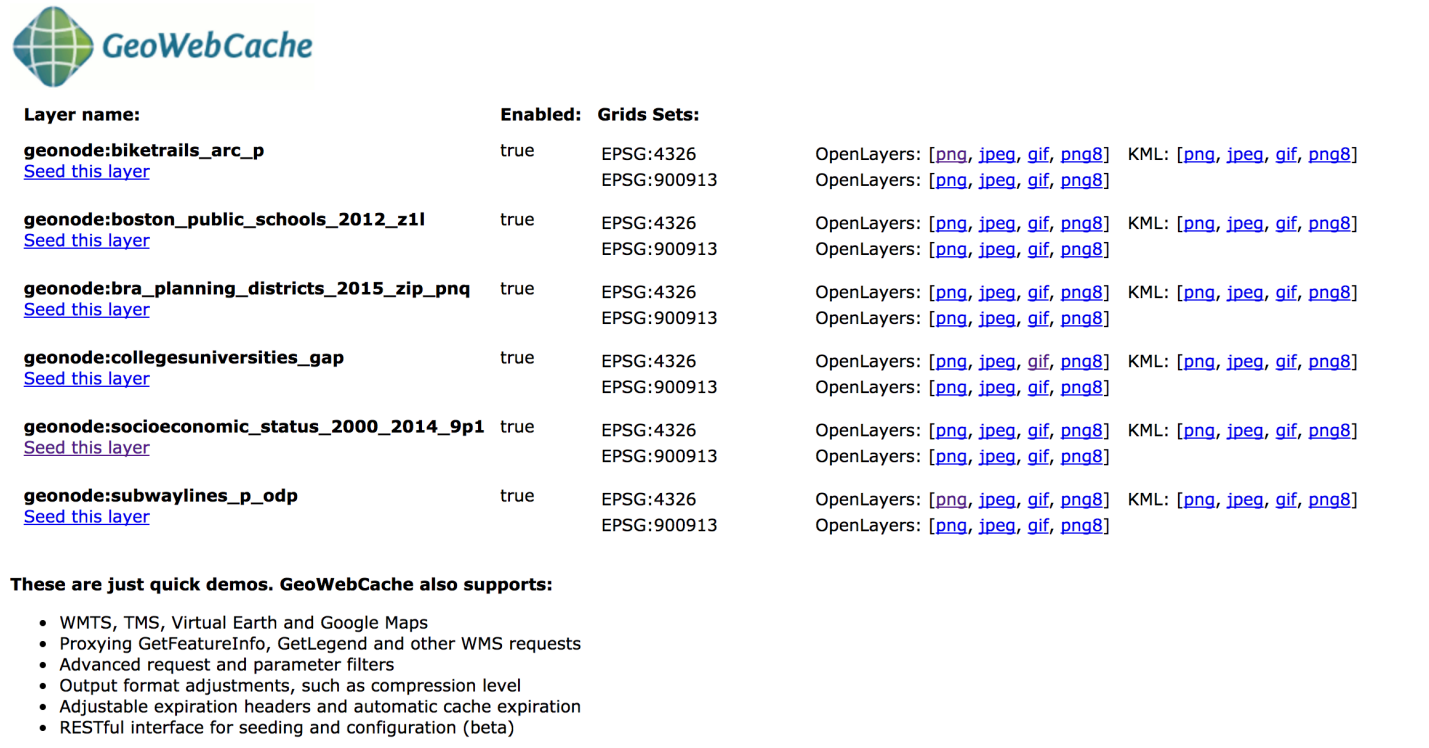
Type	Layer Name	Disk Quota	Disk Used	BlobStore	Enabled	Preview	Actions
	geonode:bra_planning_districts_2015_zip_pnq	N/A	N/A		✓	Select One	Seed/Truncate | Empty
	geonode:collegesuniversities_gap	N/A	N/A		✓	Select One	Seed/Truncate | Empty
	geonode:subwaylines_p_odp	N/A	N/A		✓	Select One	Seed/Truncate | Empty
	geonode:boston_public_schools_2012_z1l	N/A	N/A		✓	Select One	Seed/Truncate | Empty
	geonode:biketrails_arc_p	N/A	N/A		✓	Select One	Seed/Truncate | Empty
	geonode:socioeconomic_status_2000_2014_9p1	N/A	N/A		✓	Select One	Seed/Truncate | Empty

<< < 1 > >> Results 1 to 6 (out of 6 items)

From the *Tile Layers* page it is also possible to add (or remove) layers to the cache.

GWC Demo Page

GWC provides also a demo page which is accessible at this url: http://localhost:8080/geoserver/gwc/demo



GeoWebCache

Layer name:	Enabled:	Grids Sets:		
geonode:biketrails_arc_p Seed this layer	true	EPSG:4326 EPSG:900913	OpenLayers: [png, jpeg, gif, png8] OpenLayers: [png, jpeg, gif, png8]	KML: [png, jpeg, gif, png8]
geonode:boston_public_schools_2012_z1l Seed this layer	true	EPSG:4326 EPSG:900913	OpenLayers: [png, jpeg, gif, png8] OpenLayers: [png, jpeg, gif, png8]	KML: [png, jpeg, gif, png8]
geonode:bra_planning_districts_2015_zip_pnq Seed this layer	true	EPSG:4326 EPSG:900913	OpenLayers: [png, jpeg, gif, png8] OpenLayers: [png, jpeg, gif, png8]	KML: [png, jpeg, gif, png8]
geonode:collegesuniversities_gap Seed this layer	true	EPSG:4326 EPSG:900913	OpenLayers: [png, jpeg, gif, png8] OpenLayers: [png, jpeg, gif, png8]	KML: [png, jpeg, gif, png8]
geonode:socioeconomic_status_2000_2014_9p1 Seed this layer	true	EPSG:4326 EPSG:900913	OpenLayers: [png, jpeg, gif, png8] OpenLayers: [png, jpeg, gif, png8]	KML: [png, jpeg, gif, png8]
geonode:subwaylines_p_odp Seed this layer	true	EPSG:4326 EPSG:900913	OpenLayers: [png, jpeg, gif, png8] OpenLayers: [png, jpeg, gif, png8]	KML: [png, jpeg, gif, png8]

These are just quick demos. GeoWebCache also supports:

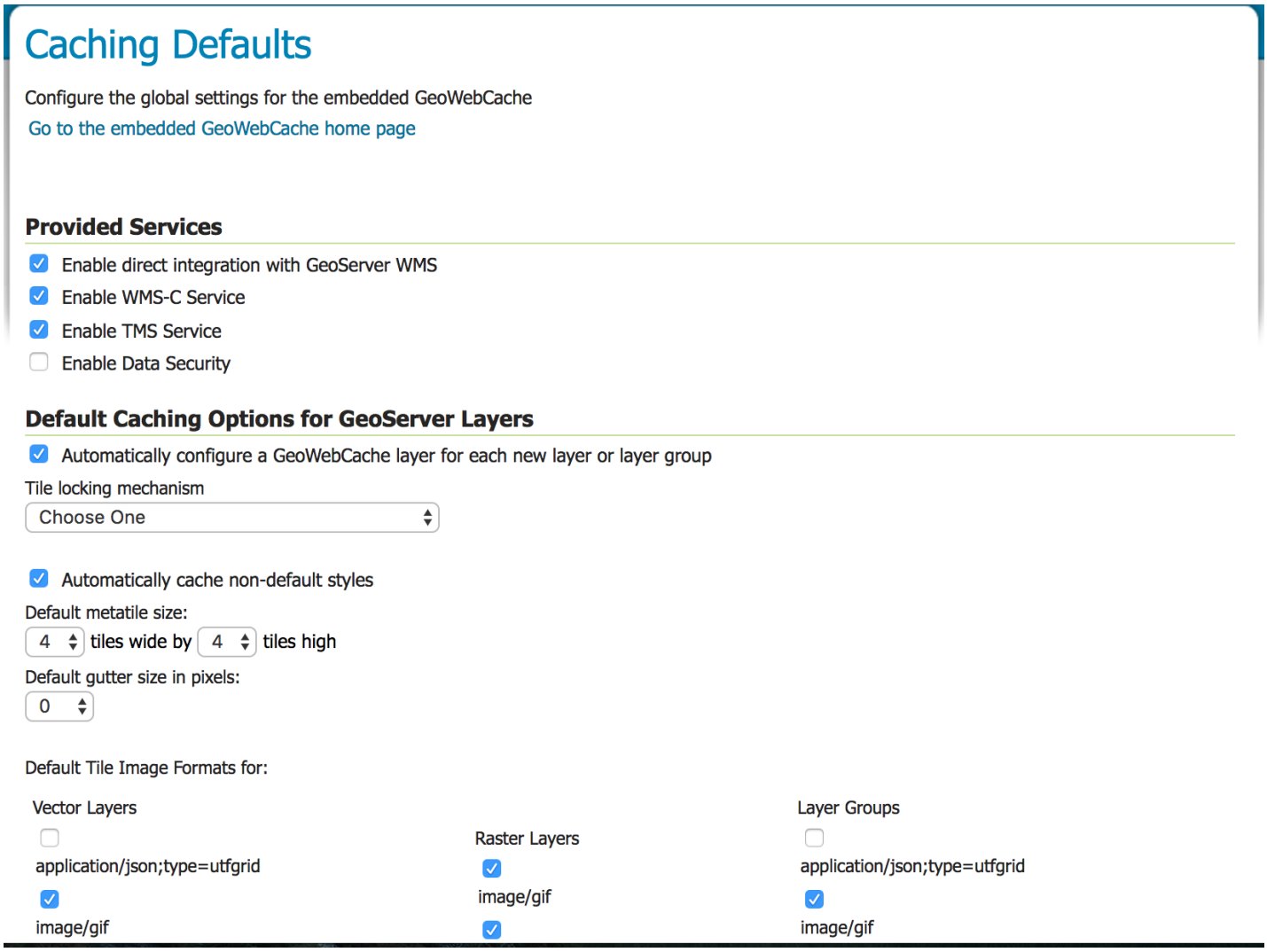
- WMTS, TMS, Virtual Earth and Google Maps
- Proxying GetFeatureInfo, GetLegend and other WMS requests
- Advanced request and parameter filters
- Output format adjustments, such as compression level
- Adjustable expiration headers and automatic cache expiration
- RESTful interface for seeding and configuration (beta)

Caching Defaults

By using the *Caching Defaults* pageit is possible:

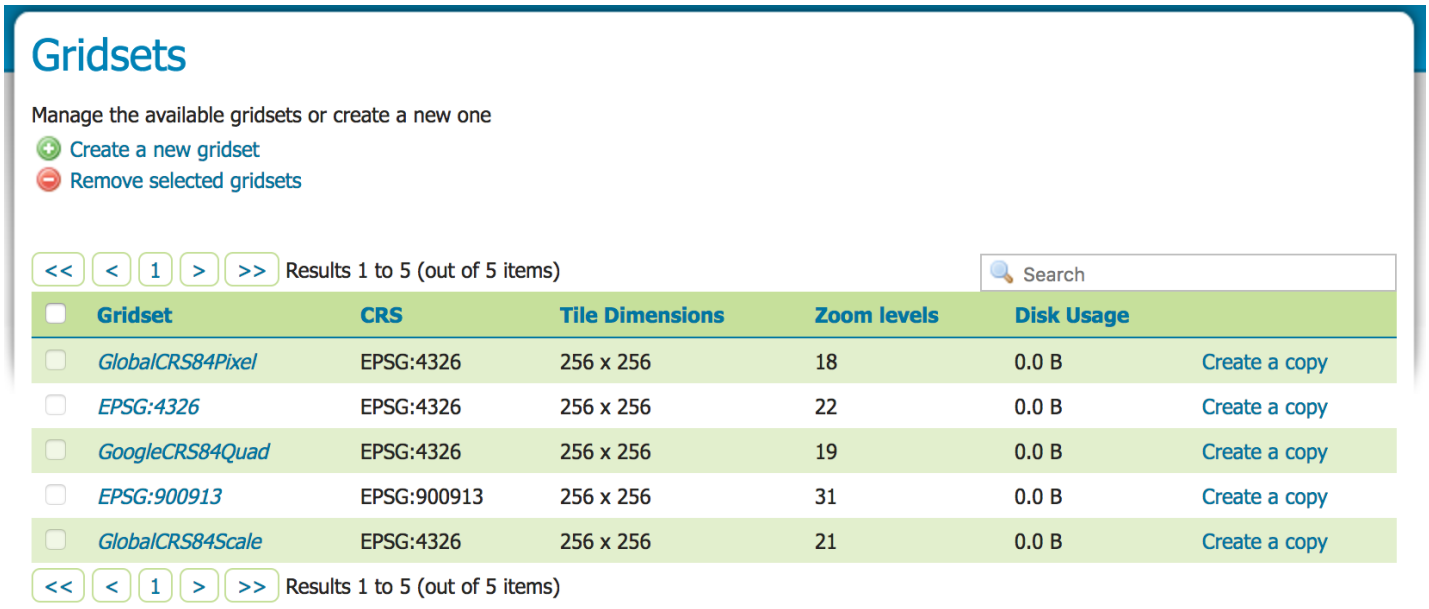
- enable direct integration with GeoServer WMS. As you have read before, by default it is enabled in GeoNode
- enable GWC Cached Web Map Service (WMS-C) endpoint: http://localhost:8080/geoserver/gwc/service/wms?
- enable GWC Tiled Map Service (TMS) endpoint: http://localhost:8080/geoserver/gwc/service/tms/1.0.0?
- enable GWC Web Map Tiled Service (WMTS) endpoint: http://localhost:8080/geoserver/gwc/service/wmts?
- set default and global options for the tile cache in GeoServer: from here it is possible to automatically configure a GWC layer for each new layer and automatically cache non-default styles (both this option are enabled by default in GeoNode)
- set default image formats that can be cached - which can be different for vector, raster and layer groups
- set the gridsets that will be automatically configured for cached layers. By default only two ones are enabled, which correspon to the most common and universal cases:
  - EPSG:4326 (geographic) with 22 maximum zoom levels and 256x256 pixel tiles
  - EPSG:900913 (spherical Mercator) with 31 maximum zoom levels and 256x256 pixel tiles

You can access WMS-C (used by GeoNode), TMS and WMTS GetCapabilities documents from the GeoServer administrative home page.



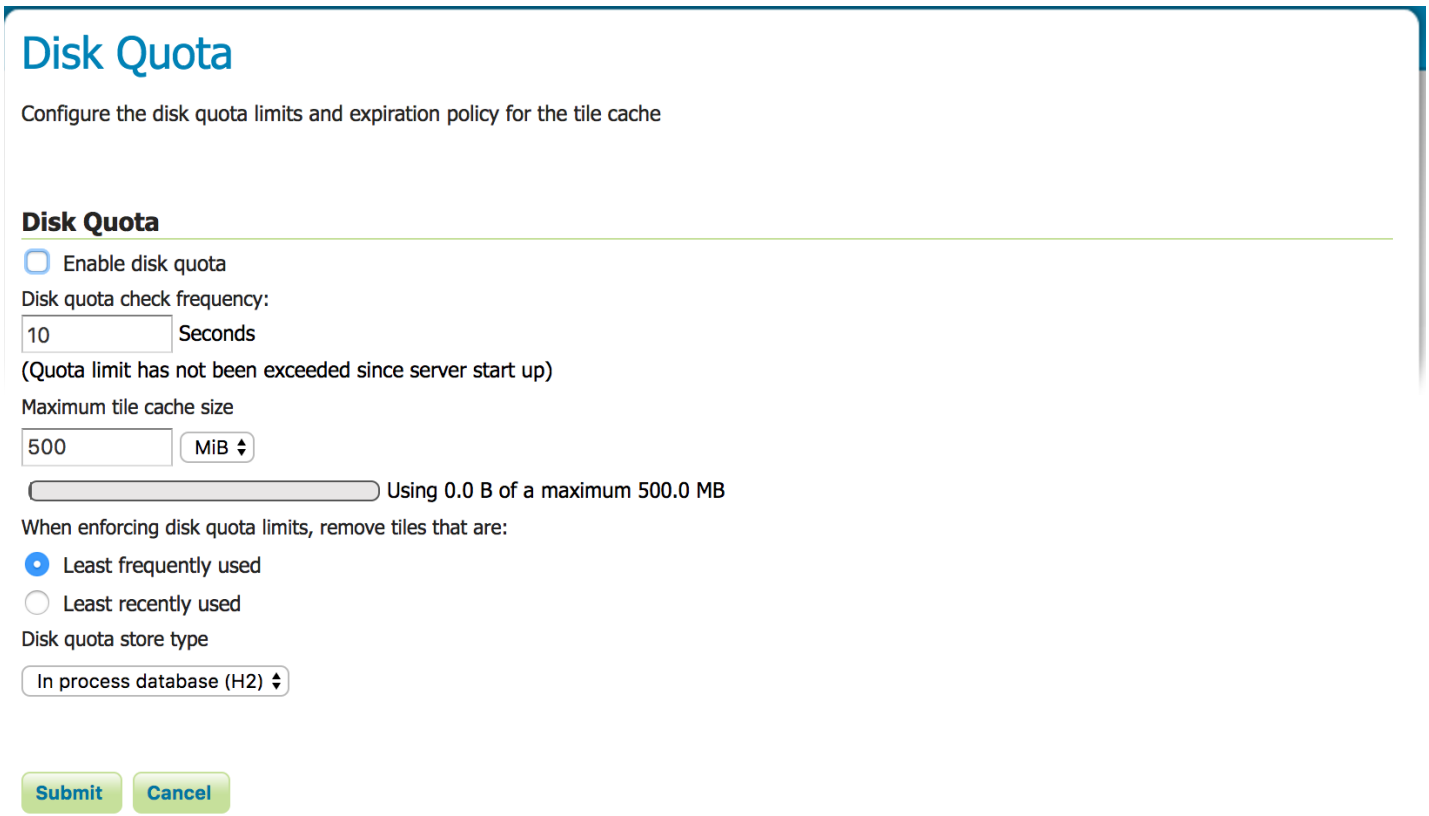
Gridsets

A gridset defines a spatial reference, bounding box, a list of zoom levels and tile dimensions. It is possible to handle gridset from the *Gridsets* page. By default there are five preconfiguret gridsets. From this section it is possible to edit, remove or create new gridsets.



Disk Quotas

The *Disk Quotas* page can be used to manage the disk usage for cached tiles - which could easily grow intensively - and allows to set the global disk space which GWC can use for its cache. By enabling it (by default it is not in GeoNode) you can specify the maximum tile cache size (by default 500 MB) and the tile removal policy (by default “Least frequently used” tiles).



GeoWebCache directory

GeoNode stores the GeoServer GWC cache in the *gwc* directory on the root of the GeoServer data directory, which is in */workshop/geonode/geoserver/data*. Both the location of the GeoServer data directory and the location of the GWC cache directory can be changed in GeoNode from the GeoServer main *web.xml* configuration file (which is, for the case of this workshop, in */workshop/geonode/geoserver/geoserver/WEB-INF/web.xml*).

Take some minutes to inspect the GWC cache directory:

```
$ cd /workshop/geonode/geoserver/data/gwc
$ ls
geonode_biketrails_arc_p                      geonode_subwaylines_p_odp
geonode_boston_public_schools_2012_z1l        geowebcache.xml
geonode_bra_planning_districts_2015_zip_pnq   _gwc_in_progress_deletes_
geonode_collegesuniversities_gap              tmp
geonode_socioeconomic_status_2000_2014_9p1
```

You can recognize one directory for each cached layer. It you inspect one of the cached layer directory you should recognize a structure like this (you may need to install the *tree* Linux command running 'sudo apt install tree' to see this output in your shell):

```
$ tree geonode_biketrails_arc_p
├── EPSG_900913_02
│   └── 0_0
│       └── 01_02.png
├── EPSG_900913_06
│   └── 01_02
│       ├── 0018_0040.png
│       └── 0019_0040.png
├── EPSG_900913_07
│   └── 02_05
│       ├── 0037_0080.png
│       ├── 0038_0080.png
│       └── 0039_0080.png
├── EPSG_900913_08
│   └── 02_05
│       ├── 0075_0160.png
│       ├── 0075_0161.png
│       ├── 0076_0160.png
│       ├── 0076_0161.png
│       ├── 0077_0160.png
│       ├── 0077_0161.png
│       ├── 0078_0160.png
│       └── 0078_0161.png
├── EPSG_900913_09
│   └── 04_10
│       ├── 0151_0320.png
│       ├── 0151_0321.png
```

For each supported spatial reference system (EPSG:900913 and EPSG:4326 when using GeoNode), zoom level and style GWC create a different directory where the tiles are stored.

Try to add another style to the layer to see the difference.

Requests to the GWC endpoints

WMS-C

The WMS Tiling Client Recommendation, or WMS-C for short, is a recommendation set by OSGeo for making tiled requests using WMS.

If a layer is configured to be cached, as it happens by default in GeoNode, the mapping client is sending GetMap request to the WMS-C endpoint using the same GetMap parameters of a GetMap request to the WMS. The only thing to change is the TILED=true parameter appended to the request:

http://localhost:8080/geoserver/wms?LAYERS=geonode%3Asocioeconomic_status_2000_2014_9p1&TRANSPARENT=TRUE&SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&STYLES=&FORMAT=image%2Fpng&TILED=true&SRS=EPSG%3A900913&BBOX=-7912761.16698,5209947.8471924,-7910315.1820752,5212393.8320972&WIDTH=256&HEIGHT=256

TMS

For an explanation of Tile Map Service Specification (TMS) check here.

z is the zoom level, while x and y define a given tile for a given zoom level.

Request to the TMS endpoints can be run in this format:

http://localhost:8080/geoserver/gwc/service/tms/1.0.0/layername/z/x/y.formatExtension

For example:

http://localhost:8080/geoserver/gwc/service/tms/1.0.0/geonode:biketrails_arc_p/10/610/750.png

In order to support multiple formats and spatial reference systems, the general path is:

http://localhost:8080/geoserver/gwc/tms/1.0.0/layername@grisetId@formatExtension/z/x/y.format

WMTS

WMTS is the OGC Web Map Tile Service Standard.

Here is how to make a GetTile request to the WMTS endpoint in GeoServer:

http://localhost:8080/geoserver/gwc/service/wmts?SERVICE=WMTS&REQUEST=GetTile&VERSION=1.0.0&LAYER=geonode:biketrails_arc_p&STYLE=_null&TILEMATRIXSET=EPSG%253A900913&TILEMATRIX=EPSG%253A900913%253A3&TILEROW=2&TILECOL=2&FORMAT=image%252Fpng

While OWSLib does not support WMS-C and TMS, you can use OWSLib to interact with WMTS:

```
>>> from owslib.wmts import WebMapTileService
>>> wmts = WebMapTileService('http://localhost:8080/geoserver/gwc/service/wmts?')
>>> print wmts.identification.title
Web Map Tile Service - GeoWebCache
>>> print wmts.identification.abstract

>>> for operation in wmts.operations:
        print operation.name
GetCapabilities
GetTile
GetFeatureInfo

>>> list(wmts.contents)
['geonode:boston_public_schools_2012_z1l',
 'geonode:biketrails_arc_p',
 'geonode:socioeconomic_status_2000_2014_9p1',
 'geonode:subwaylines_p_odp']

>>> layer = wmts['geonode:biketrails_arc_p']
>>> print layer.title
Bike Trails updated
>>> print layer.name
geonode:biketrails_arc_p
>>> print layer.abstract
2009 MBTA bike trails updated
```

GeoWebCache REST API

As GeoServer, GeoWebCache provides a REST API for working programmatically with the GeoWebCache configuration.

The GWC REST API endpoint is at: http://localhost:8080/geoserver/gwc/rest

By using the GWC REST API it is possible to:

- Add, modify or delete layers from GWC
- Seeding and truncating the layer's cache
- Changing the Disk Quotas configuration

One common use case it is to use the REST API to refresh the cache for a layer when it is edited from clients which are not using WFS-T (when using WFS-T the cache for the layer it is automatically udpated)

foss4g_2017_geonode_solr

Catalogue services with pycsw

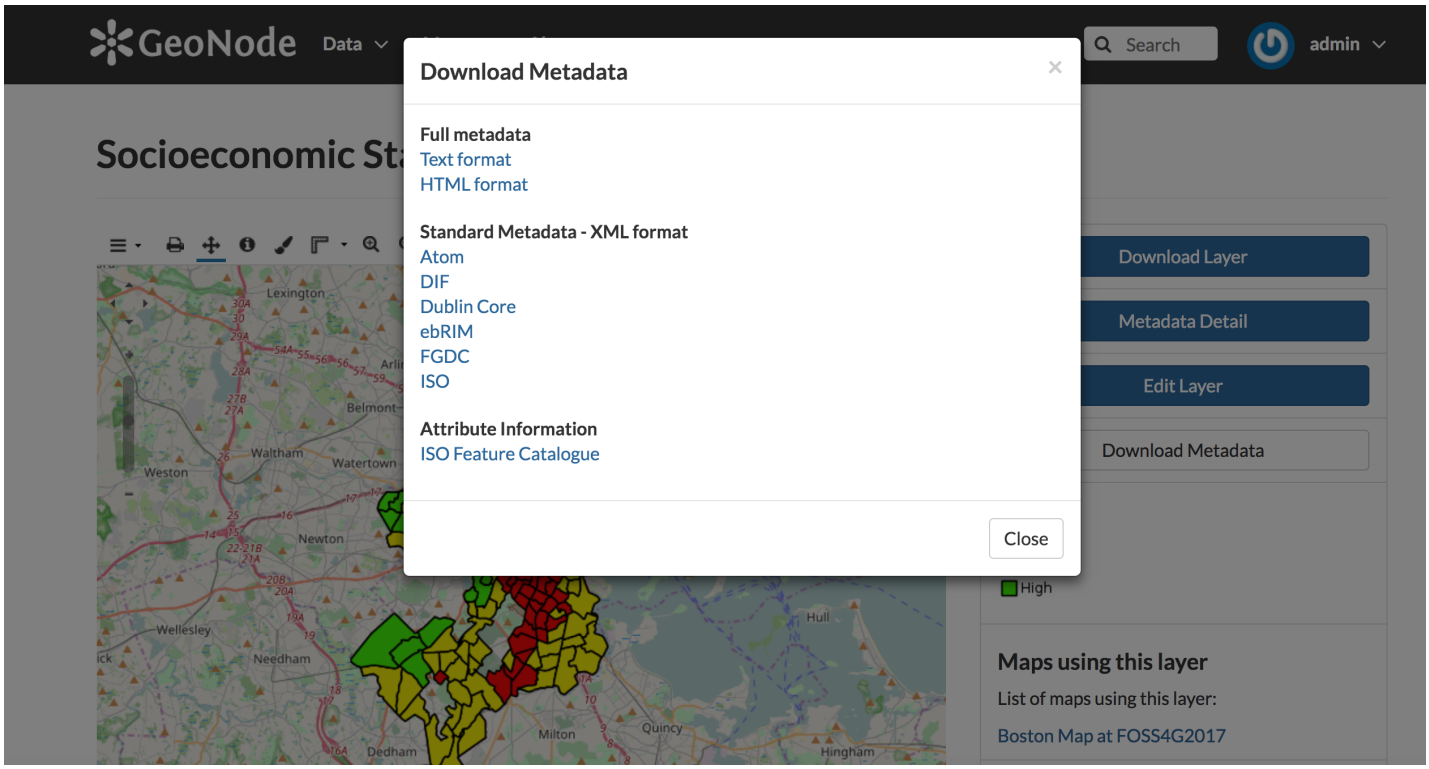
In GeoNode OGC Catalogue Service for the Web, CSW services are provided by pycsw.

> pycsw is an OGC CSW server implementation written in Python. Started in 2010 (more formally announced in 2011), pycsw allows for the publishing and discovery of geospatial metadata via numerous APIs (CSW 2/CSW 3, OpenSearch, OAI-PMH, SRU), providing a standards-based metadata and catalogue component of spatial data infrastructures. pycsw is Open Source, released under an MIT license, and runs on all major platforms (Windows, Linux, Mac OS X).

CSW is a standard for exposing a catalogue of geospatial entities on HTTP. In a GeoNode SDI or portal CSW endpoints are provided by pycsw, which is an underlying component of the GeoNode's stack. Alternatively, if needed, it is possible to replace pycsw with GeoNetwork.

Using pycsw in GeoNode

In GeoNode you can easily access to the CSW record for one layer by clicking in the *Download Metadata* button in the layer page. A form will pop up and you will be able to access to metadata provided by pycsw in a series of different formats (Atom, Dublin, FGDC, Text, HTML and many others)



For example, by clicking on the *ISO* link you will access to the metadata for the layer in the ISO format, which corresponds to this *GetRecordById* request in pycsw:

http://localhost:8000/catalogue/csw?outputschema=http%3A%2F%2Fwww.isotc211.org%2F2005%2Fgmd&service=CSW&request=GetRecordById&version=2.0.2&elementsetname=full&id=8bcf5bfc-5cfc-11e7-8103-02d8e4477a33

Look at the pycsw output of this request, and you should recognize the information which you filled in an earlier tutorial:

- title
- keywords
- place keywords
- category

You can also notice other information which was generated by GeoNode behind the scenes when the layer has been uploaded:

- identifier of the layer, which uniquely identifies the layer in the catalogue (note that the GetRecordById request uses this identifier to access the record)
- creation dateStamp
- spatial reference system and bounding box
- thumbnail url

- format of the resource
- several OGC endpoints

If you want to add missing metadata you can visit the layer metadata page of the layer and press on *Edit Layer* > *Edit Metadata*

pycsw operations

pycsw implements all of the CSW standard operations, including the optional ones:

- *GetCapabilities* retrieves service metadata from the server
- *DescribeRecord* allows a client to discover elements of the information model supported by the target catalog service
- *GetRecords* search for records using a series of criteria
- *GetRecordById* retrieves metadata for one record (layer) of the catalogue by its id
- *GetDomain* (optional) retrieves runtime information about the range of values of a metadata record element or request parameter
- *Harvest* (optional) create/update metadata by asking the server to 'pull' metadata from somewhere
- *Transaction* (optional) create/edit metadata by 'pushing' the metadata to the server

In the following you will have a look at some of these operations.

GetCapabilities

GetCapabilities retrieves service metadata from the server. You can try the GetCapabilities operation by going to this url: http://localhost:8000/catalogue/csw?service=CSW&version=2.0.2&request=GetCapabilities

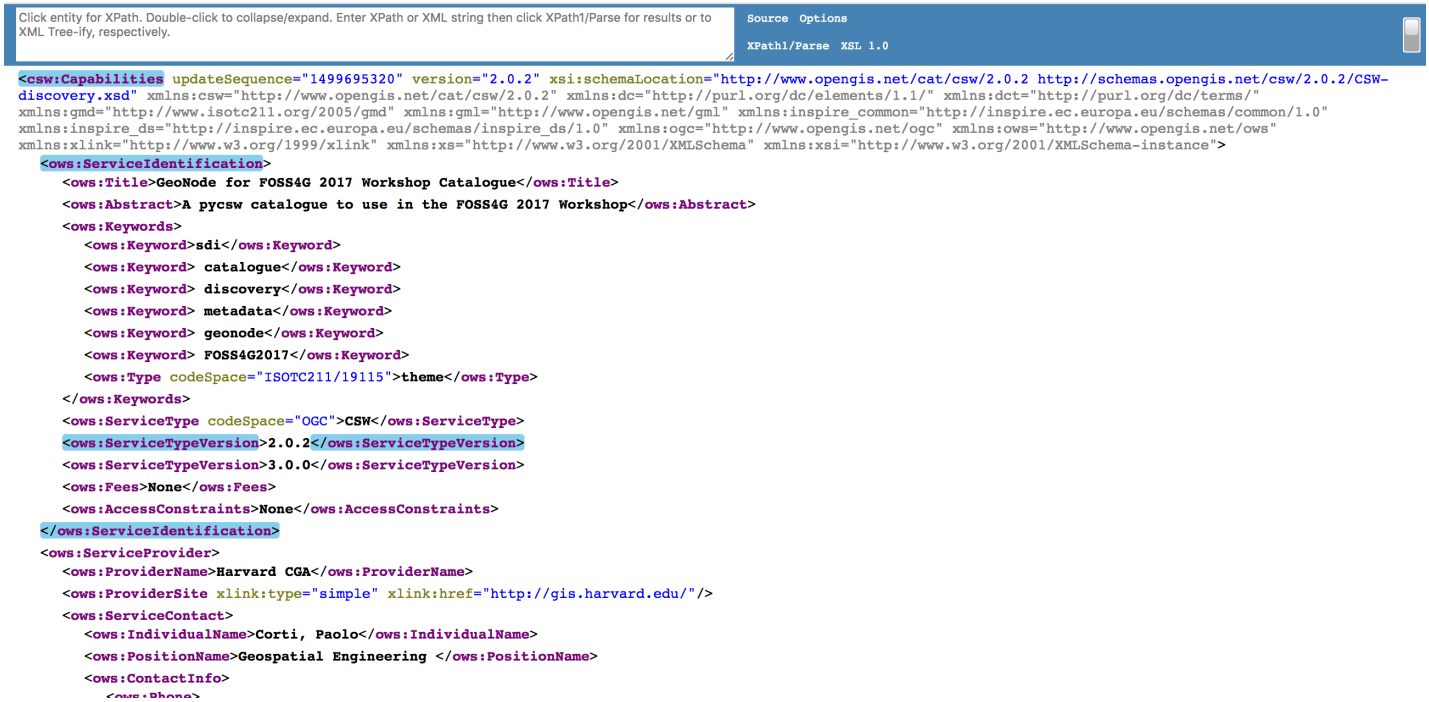
You should recognize in the pycsw GetCapabilities output the following sections:

- Service identification (title, abstract keywords...)
- Service provider (several contact information)
- Operations metadata (one for each supported operation)
- Filter capabilities

As with many other GeoNode settings, you can change the metadata of the pycsw server (service identification and provider sections) of your GeoNode instance by editing the */workshop/geonode/geonode/local_settings.py* file. Open that file (or create it if you still have not), and add the following information, changing the settings to your needs:

```
# pycsw settings
PYCSW = {
    # pycsw configuration
    'CONFIGURATION': {
        # uncomment / adjust to override server config system defaults
        # 'server': {
        #    'maxrecords': '10',
        #    'pretty_print': 'true',
        #    'federatedcatalogues': 'http://catalog.data.gov/csw'
        # },
        'metadata:main': {
            'identification_title': 'GeoNode for FOSS4G 2017 Workshop Catalogue',
            'identification_abstract': 'A pycsw catalogue to use in the FOSS4G 2017 W
            'identification_keywords': 'sdi, catalogue, discovery, metadata, geonode,
            'identification_keywords_type': 'theme',
            'identification_fees': 'None',
            'identification_accessconstraints': 'None',
            'provider_name': 'Harvard CGA',
            'provider_url': 'http://gis.harvard.edu/',
            'contact_name': 'Corti, Paolo',
            'contact_position': 'Geospatial Engineering ',
            'contact_address': '1737 Cambridge Street, K350',
            'contact_city': 'Cambridge',
            'contact_stateorprovince': 'Massachusetts',
            'contact_postalcode': '02138',
            'contact_country': 'US',
            'contact_phone': '+1-617-496-0103',
            'contact_fax': '+1-617-496-5149',
            'contact_email': 'pcorti@...',
            'contact_url': 'Contact URL',
            'contact_hours': 'Hours of Service',
            'contact_instructions': 'During hours of service. Off on ' \
            'weekends.',
            'contact_role': 'pointOfContact',
        },
        'metadata:inspire': {
            'enabled': 'true',
            'languages_supported': 'eng,gre',
            'default_language': 'eng',
            'date': 'YYYY-MM-DD',
            'gemet_keywords': 'Utility and governmental services',
            'conformity_service': 'notEvaluated',
            'contact_name': 'Organization Name',
            'contact_email': 'Email Address',
            'temp_extent': 'YYYY-MM-DD/YYYY-MM-DD',
        }
    }
}
```

Now restart GeoNode and check if the GetCapabilities response is correctly updated by pycsw



DescribeRecord

DescribeRecord allows a client to discover elements of the information model supported by the target catalog service. You can try this request by going to this url:

http://localhost:8000/catalogue/csw?service=CSW&version=2.0.2&request=DescribeRecord&TypeName=csw:Record

GetRecords

GetRecords search for records using a series of filters. This is returning all of the pycsw records:

http://localhost:8000/catalogue/csw?service=CSW&version=2.0.2&request=GetRecords&ElementSetName=full&typenames=csw:Record&resulttype=results

It is possible to retrieve records using several types of text and spatial filters. For example here is a request to filter records which have the word “bike” in metadata text and fall within a bounding box of (-180, -90, 180, 90):

http://localhost:8000/catalogue/csw?service=CSW&version=2.0.2&request=GetRecords&ElementSetName=full&typenames=csw:Record&resulttype=results&q=bike&bbox=-180,-90,180,90

A great source to get a good understanding of the filters is to look at the pycsw test suite

GetRecordById

GetRecordById retrieves metadata for one record (layer) of the catalogue by its id. You have seen this request previously, when requesting the metadata record from the GeoNode user interface:

http://localhost:8000/catalogue/csw?outputschema=http%3A%2F%2Fwww.isotc211.org%2F2005%2Fgmd&service=CSW&request=GetRecordById&version=2.0.2&elementsetname=full&id=8bcf5bfc-5cfc-11e7-8103-02d8e4477a33

Accessing the pycsw catalogue from Python

Now you will access and query the GeoNode pycsw catalogue using OWSLib (as you did with OCG services exposed by GeoServer and GeoWebCache in previous tutorials).

Go to your Django shell for this purpose. If your Django shell is not open, open a new shell, log in the vagrant box, activate the virtualenv and run again the Django shell:

```
✗ vagrant ssh
$ . /workshop/env/bin/activate
$ cd /workshop/geonode/
$ python manage.py shell
```

As a first thing you will connect to the pycsw CSW endpoint and query the service identification and provider:

```
>>> from owslib.csw import CatalogueServiceWeb
>>> csw = CatalogueServiceWeb('http://localhost:8000/catalogue/csw')
>>> print csw.identification.title
GeoNode for FOSS4G 2017 Workshop Catalogue
>>> print csw.identification.abstract
A pycsw catalogue to use in the FOSS4G 2017 Workshop
>>> print csw.identification.keywords
['sdi', ' catalogue', ' discovery', ' metadata', ' geonode', ' FOSS4G2017']
>>> print csw.provider.contact
<owslib.ows.ServiceContact object at 0x7f296e99a850>
>>> print csw.provider.name
Harvard CGA
>>> print csw.provider.url
http://gis.harvard.edu/
>>> print csw.provider.contact.name
```

```
Corti, Paolo
>>> print csw.provider.contact.city
Cambridge
```

Now print the supported CSW operations:

```
for operation in csw.operations:
    print operation.name
GetCapabilities
DescribeRecord
GetDomain
GetRecords
GetRecordById
GetRepositoryItem
```

Get all metadata records which contains the text 'boston':

```
>>> from owslib.fes import PropertyIsEqualTo
>>> boston_query = PropertyIsEqualTo('csw:AnyText', 'boston')
>>> csw.getrecords2(constraints=[boston_query])

>>> print csw.results
{'matches': 5, 'nextrecord': 0, 'returned': 5}

>>> for record in csw.records:
        print csw.records[record].title
Boston Public Schools (2012)
Socioeconomic Status (2000 - 2014)
MBTA Subway Lines
Colleges and Universities in Boston
Boston Planning Districts (BRA)
```

Get all metadata in the US extent and containing the text 'boston':

```
>>> from owslib.fes import BBox
>>> us_bbox = BBox([-125,24,-66,49])
>>> csw.getrecords2(constraints=[boston_query])
>>> csw.getrecords2(constraints=[us_bbox])
>>> print csw.results
{'matches': 1, 'nextrecord': 0, 'returned': 1}
```

Get a record by the id (make sure to use the uuid of one of the layers you uploaded):

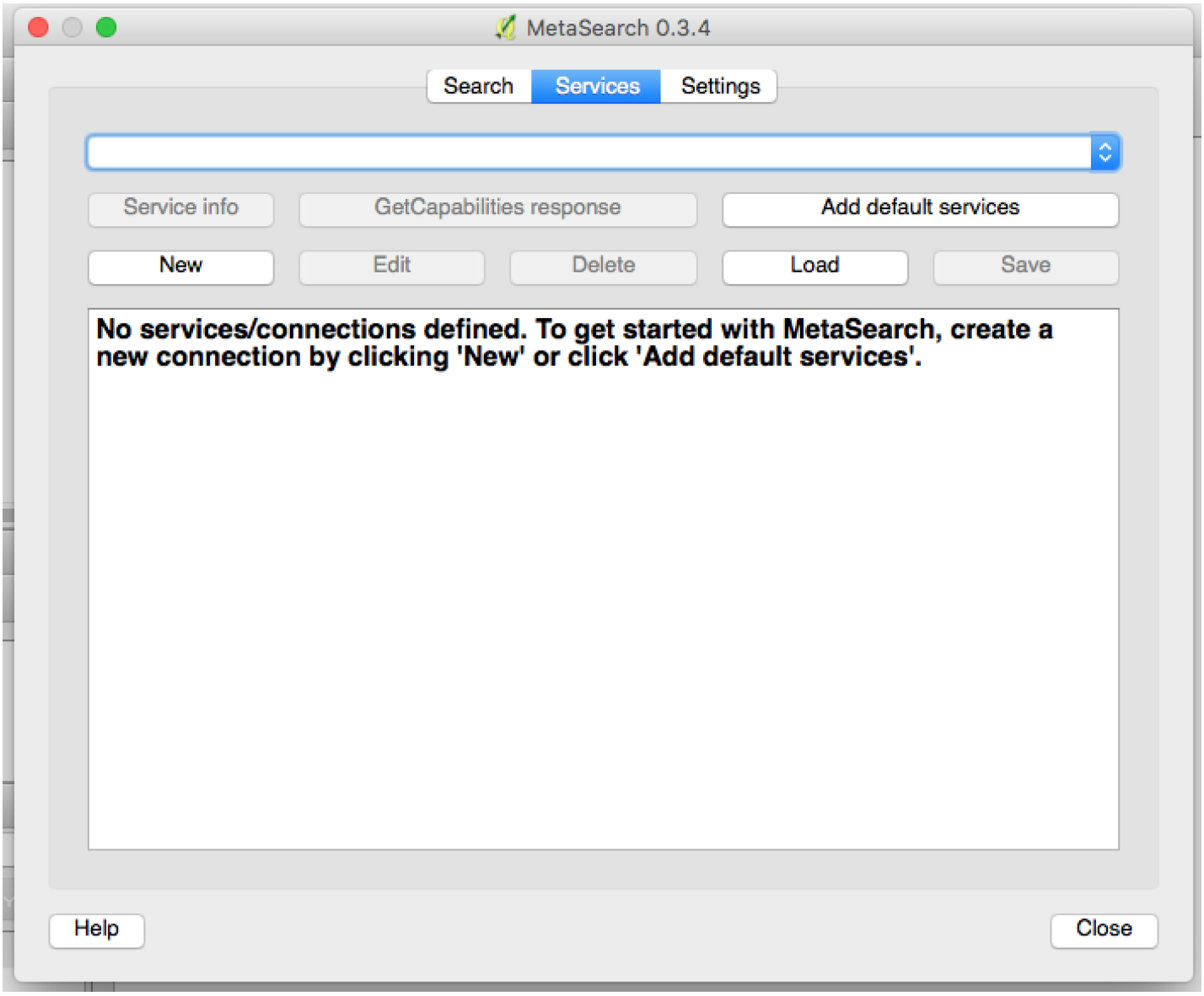
```
>>> csw.getrecordbyid(id=['8bcf5bfc-5cfc-11e7-8103-02d8e4477a33'])
>>> csw.records
```

```
    OrderedDict([('8bcf5bfc-5cfc-11e7-8103-02d8e4477a33',
                  <owslib.csw.CswRecord at 0x7f296e97d990>)])
>>> record = csw.records['8bcf5bfc-5cfc-11e7-8103-02d8e4477a33']
>>> print record.title
Socioeconomic Status (2000 - 2014)
>>> print record.abstract
Socio-economic Status Index Range
```

Accessing the pycsw catalogue from QGIS

Now you will access and query the pycsw GeoNode catalogue from within the QGIS desktop client. The screenshots here are generated using QGIS 2.14 LTR, but following versions should be similar.

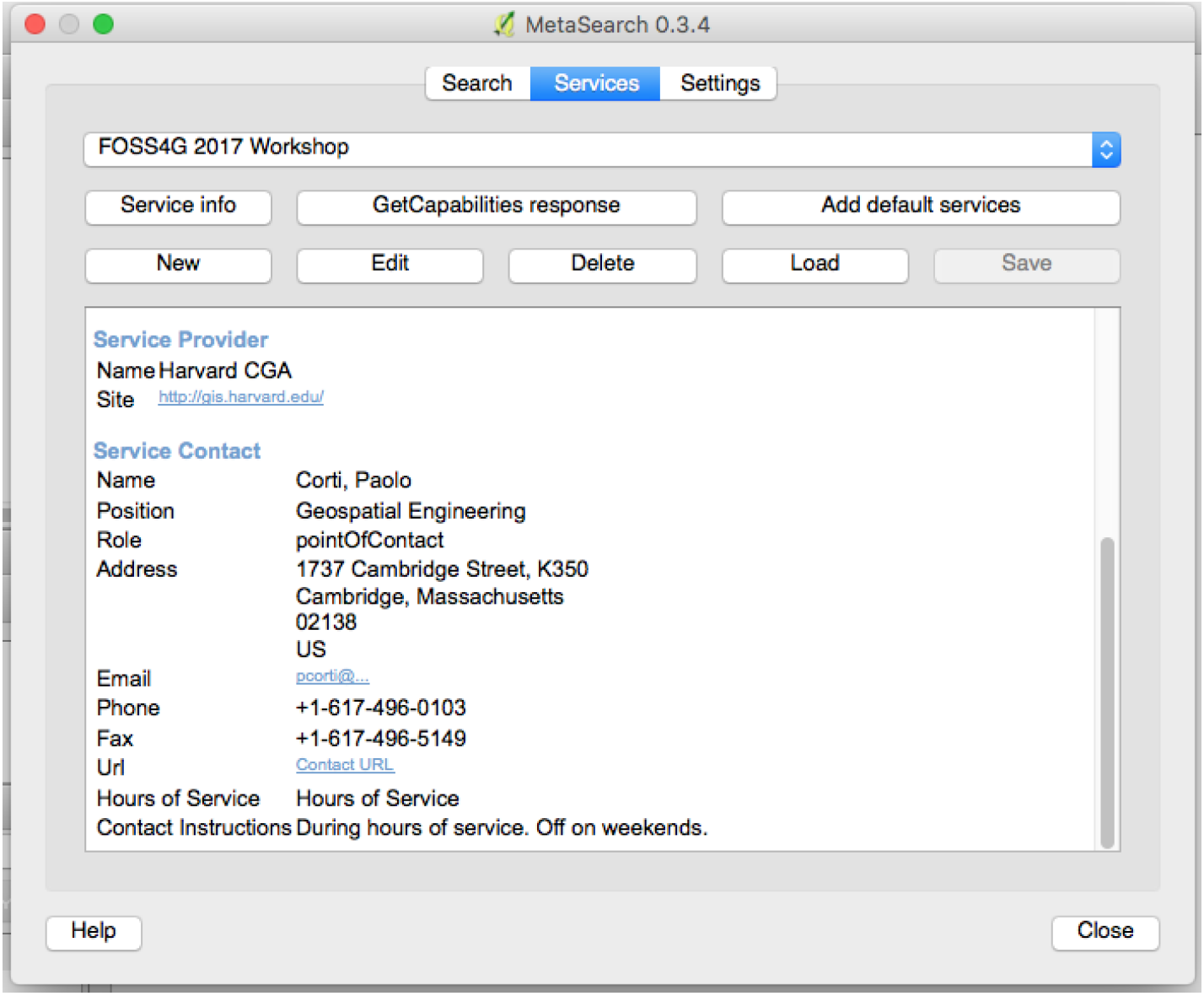
As a first thing, click on the *Web* menu, and then on *MetaSearch* > *MetaSearch*



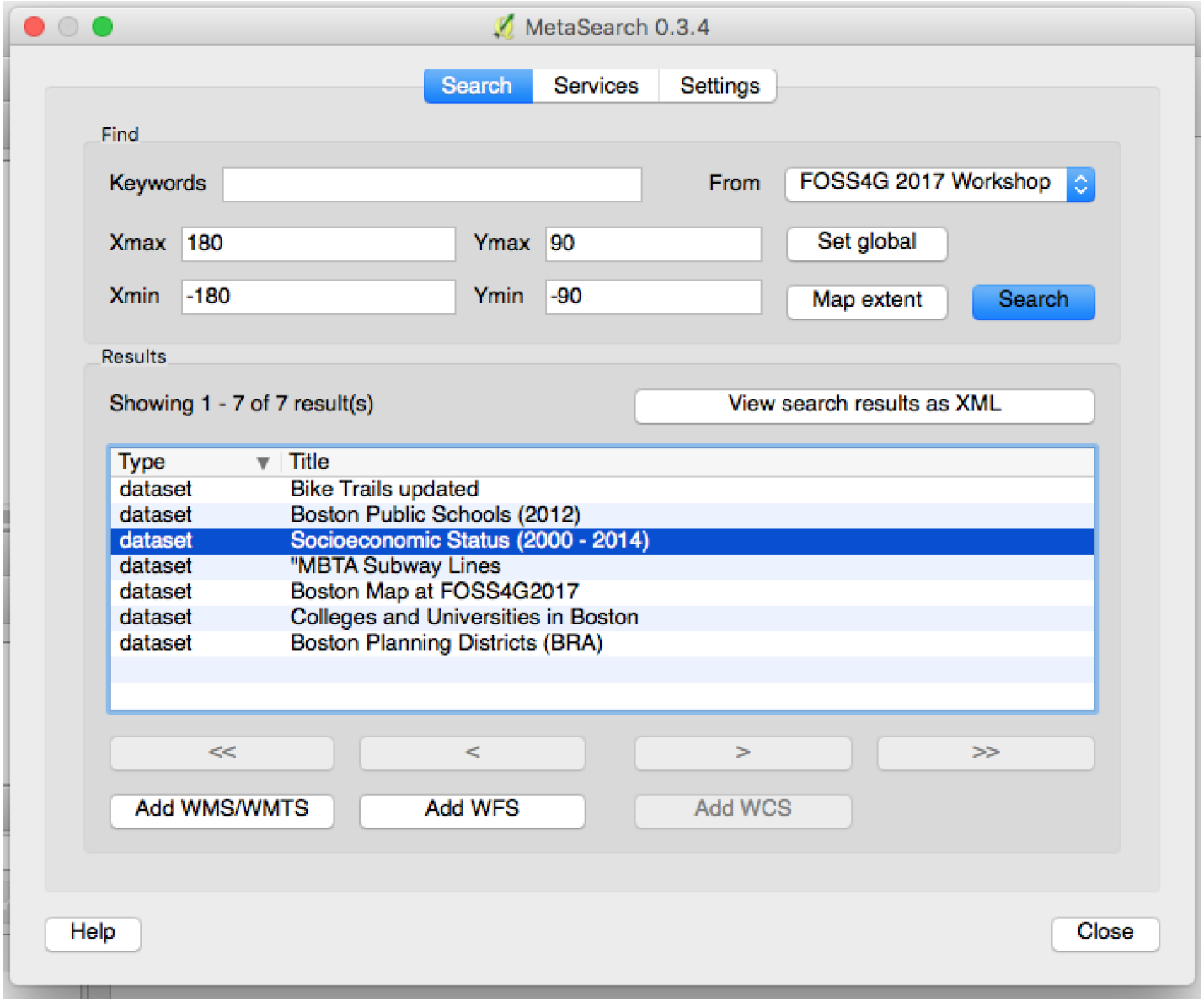
Now click on the *New* button, and compile the form as following:

- name: FOSS4G 2017 Workshop
- url: http://localhost:8000/catalogue/csw?

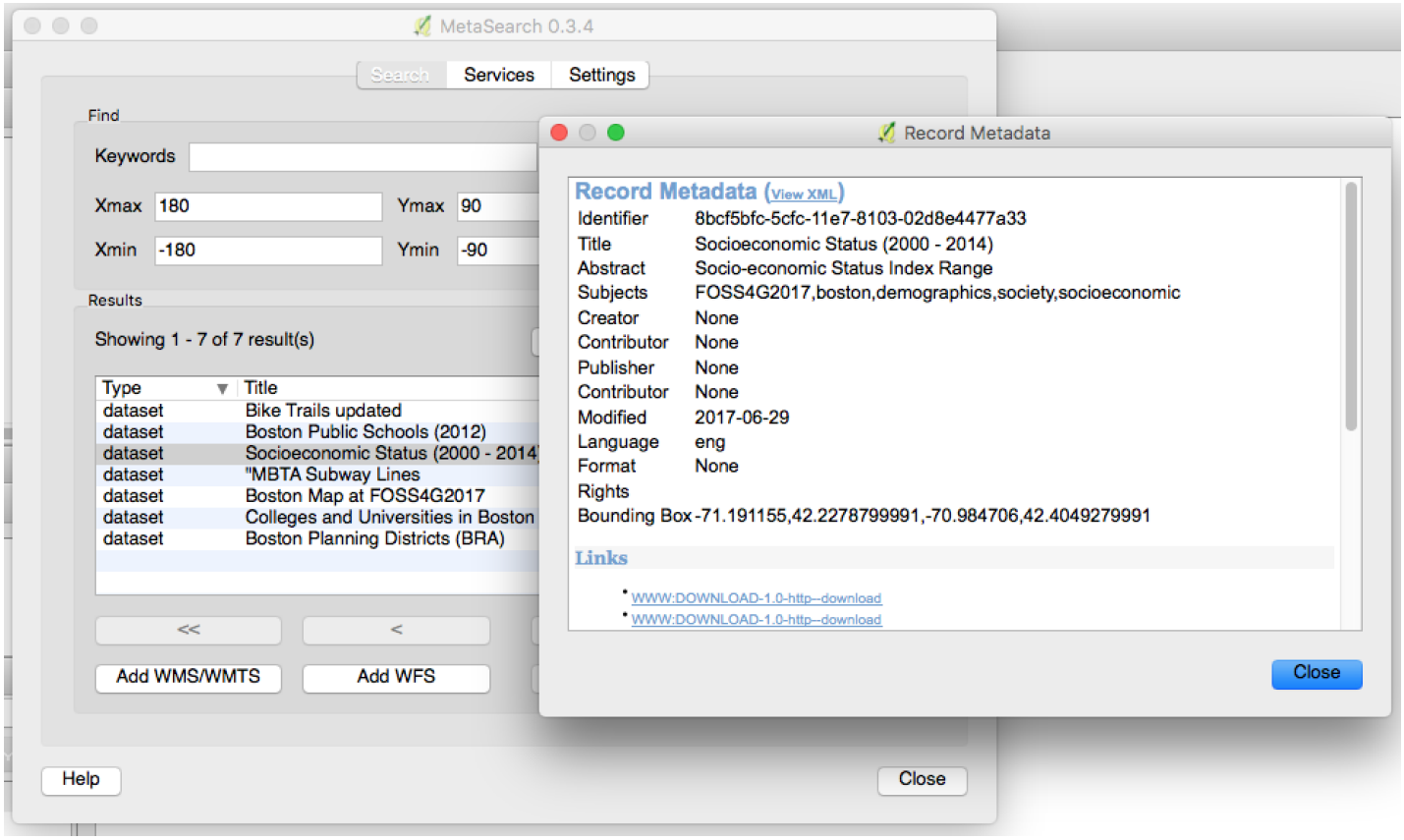
Click on the *Service Info* button, and the identification of the CSW service should appear in the MetaSearch main window:



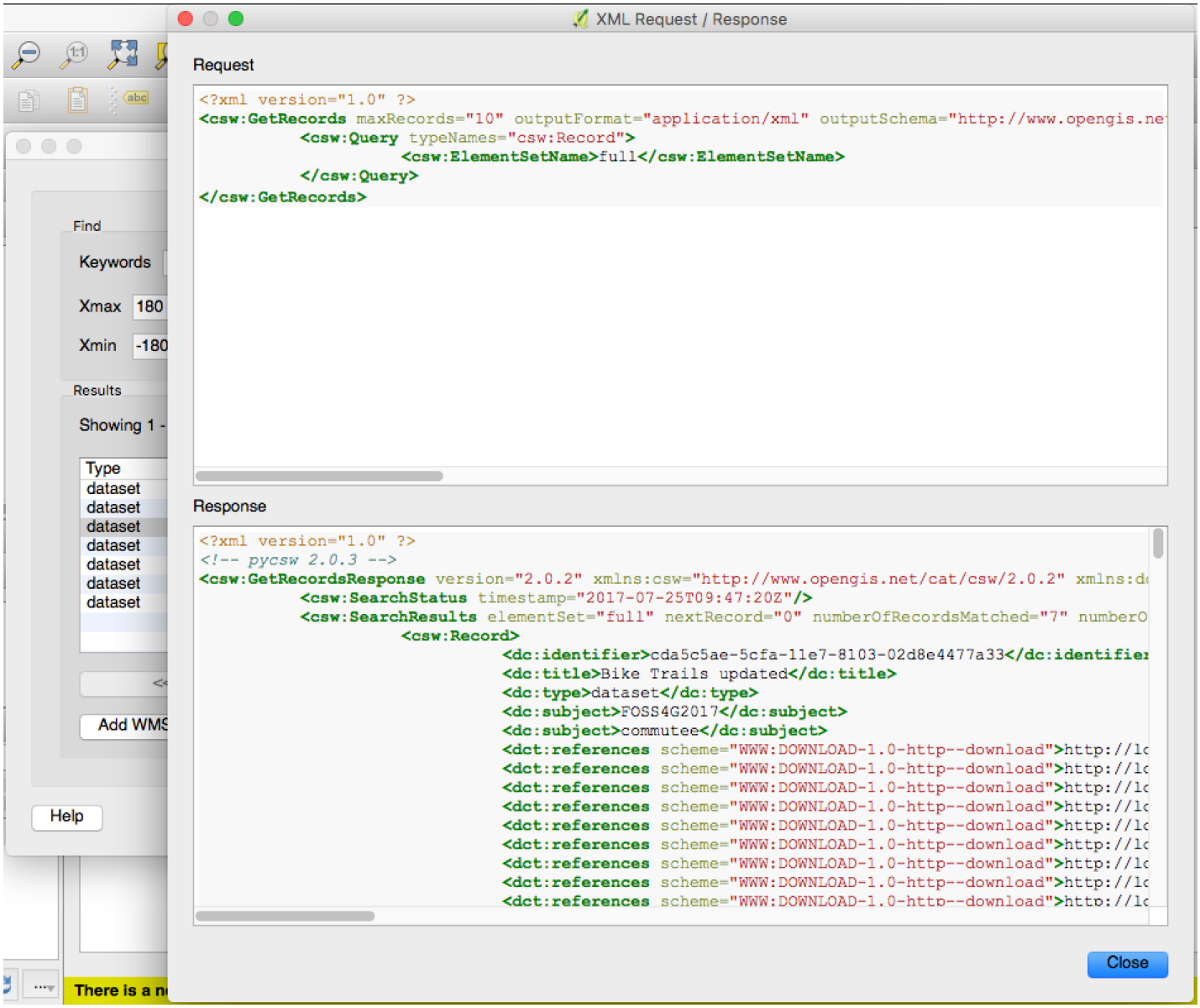
Now, click in the *Search* tab, and try a basic search by clicking on the *Search* button. All of the pycsw records should show:



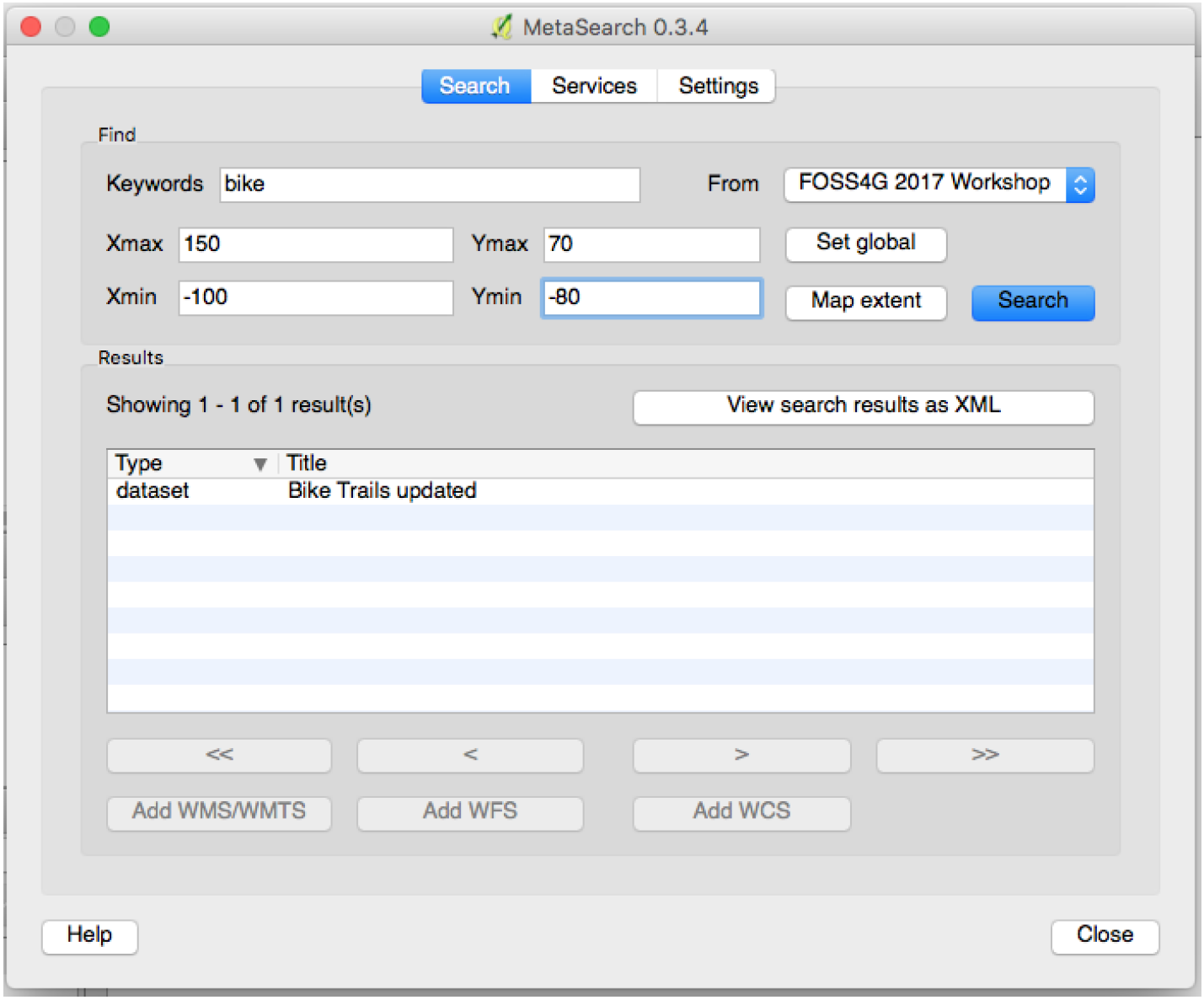
If you double click on one of the layer you should see the record information



You can also see the GetRecords request performed to pycsw by clicking on the *View search results as xml* button



Try playing inserting different extent parameters and text to perform more advanced queries


MetaSearch 0.3.4
Search
Services
Settings
Find
Keywords
bike
From
FOSS4G 2017 Workshop
Xmax
150
Ymax
70
Set global
Xmin
-100
Ymin
-80
Map extent
Search
Results
Showing 1 - 1 of 1 result(s)
View search results as XML
Type
Title
dataset
Bike Trails updated
<<
<
>
>>
Add WMS/WMTS
Add WFS
Add WCS
Help
Close

foss4g_2017_geonode_solr

Spatial query with PostGIS

> PostgreSQL, often simply Postgres, is an open source object-relational database management system (ORDBMS) with an emphasis on extensibility and standards compliance. As a database server, its primary functions are to store data securely and return that data in response to requests from other software applications. PostgreSQL is ACID-compliant and transactional. PostgreSQL has updatable views and materialized views, triggers, foreign keys; supports functions and stored procedures, and other expandability.

> PostGIS is an open source software program that adds support for geographic objects to the PostgreSQL object-relational database. PostGIS follows the Simple Features for SQL specification from the Open Geospatial Consortium (OGC).

There are several good reasons to store the Django database (which by default in your setup uses sqlite) and the vector data uploads database using PostgreSQL and PostGIS. Most notably:

- advanced security and multiuser
- data integrity
- better performances in GeoServer, specially with complex styles
- support for editing layers
- support for spatial SQL

In this step of the workshop you will configure GeoNode to use PostgreSQL/PostGIS for users uploaded layers. You will still keep the Django database in sqlite

Bonus step: uses PostgreSQL for the Django database

Database creation

Login with the *postgres* user and open **psql**, the PostgreSQL command line utility. If you want to get more acquainted to use psql you can check out its documentation

```
$ sudo su postgres
postgres@ubuntu-xenial:/workshop$ psql
```

```
psql (9.5.7)
Type "help" for help.
```

Create a *geonode* user (role), with the password set to *geonode*. Then create a database, named *geonode* owned by the geonode user: this database will be used by GeoNode to store any shapefile uploaded to it

```
postgres=# CREATE ROLE geonode WITH SUPERUSER LOGIN PASSWORD 'geonode';
CREATE ROLE
postgres=# CREATE DATABASE geonode WITH OWNER geonode;
CREATE DATABASE
```

Now connect to the *geonode* database and install the PostGIS extension. Then exit psql

```
postgres=# \c geonode
You are now connected to database "geonode" as user "postgres".
geonode=# CREATE EXTENSION postgis;
CREATE EXTENSION
geonode=# \q
exit
```

Make sure you can connect to the *geonode* database using the *geonode* user:

```
$ psql -h localhost -U geonode geonode
Password for user geonode:
psql (9.5.7)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, co
Type "help" for help.

geonode=#
```

Configure GeoNode to use PostgreSQL/PostGIS to store uploaded vector layers:

Now you need to instruct GeoNode to use the *geonode* PostGIS database you just created as a datastore for new uploads.

Open the *local_settings.py* file you previously created and copy and paste the following lines at the beginning of it. Then save the file

```
import os

import dj_database_url

PROJECT_ROOT = os.path.abspath(os.path.dirname(__file__))

DATABASE_URL = 'sqlite:///{path}'.format(path=os.path.join(PROJECT_ROOT, 'development
DATASTORE_URL = 'postgis://geonode:geonode@localhost:5432/geonode'

DATABASES = {
    'default': dj_database_url.parse(DATABASE_URL, conn_max_age=600),
    'datastore': dj_database_url.parse(DATASTORE_URL, conn_max_age=600),
}

GEOSERVER_LOCATION = os.getenv(
    'GEOSERVER_LOCATION', 'http://localhost:8080/geoserver/'
)
GEOSERVER_PUBLIC_LOCATION = os.getenv(
    'GEOSERVER_PUBLIC_LOCATION', 'http://localhost:8080/geoserver/'
)

# OGC (WMS/WFS/WCS) Server Settings
OGC_SERVER = {
    'default': {
        'BACKEND': 'geonode.geoserver',
        'LOCATION': GEOSERVER_LOCATION,
        'LOGIN_ENDPOINT': 'j_spring_oauth2_geonode_login',
        'LOGOUT_ENDPOINT': 'j_spring_oauth2_geonode_logout',
        # PUBLIC_LOCATION needs to be kept like this because in dev mode
        # the proxy won't work and the integration tests will fail
        # the entire block has to be overridden in the local_settings
        'PUBLIC_LOCATION': GEOSERVER_PUBLIC_LOCATION,
        'USER' : 'admin',
        'PASSWORD' : 'geoserver',
        'MAPFISH_PRINT_ENABLED' : True,
        'PRINT_NG_ENABLED' : True,
        'GEONODE_SECURITY_ENABLED' : True,
        'GEOGIG_ENABLED' : False,
        'WMST_ENABLED' : False,
        'BACKEND_WRITE_ENABLED': True,
        'WPS_ENABLED' : False,
        'LOG_FILE': '%s/geoserver/data/logs/geoserver.log' % os.path.abspath(os.path.
        # Set to dictionary identifier of database containing spatial data in DATABAS
        'DATASTORE': 'datastore',
    }
}
```

Upload data to PostgreSQL/PostGIS

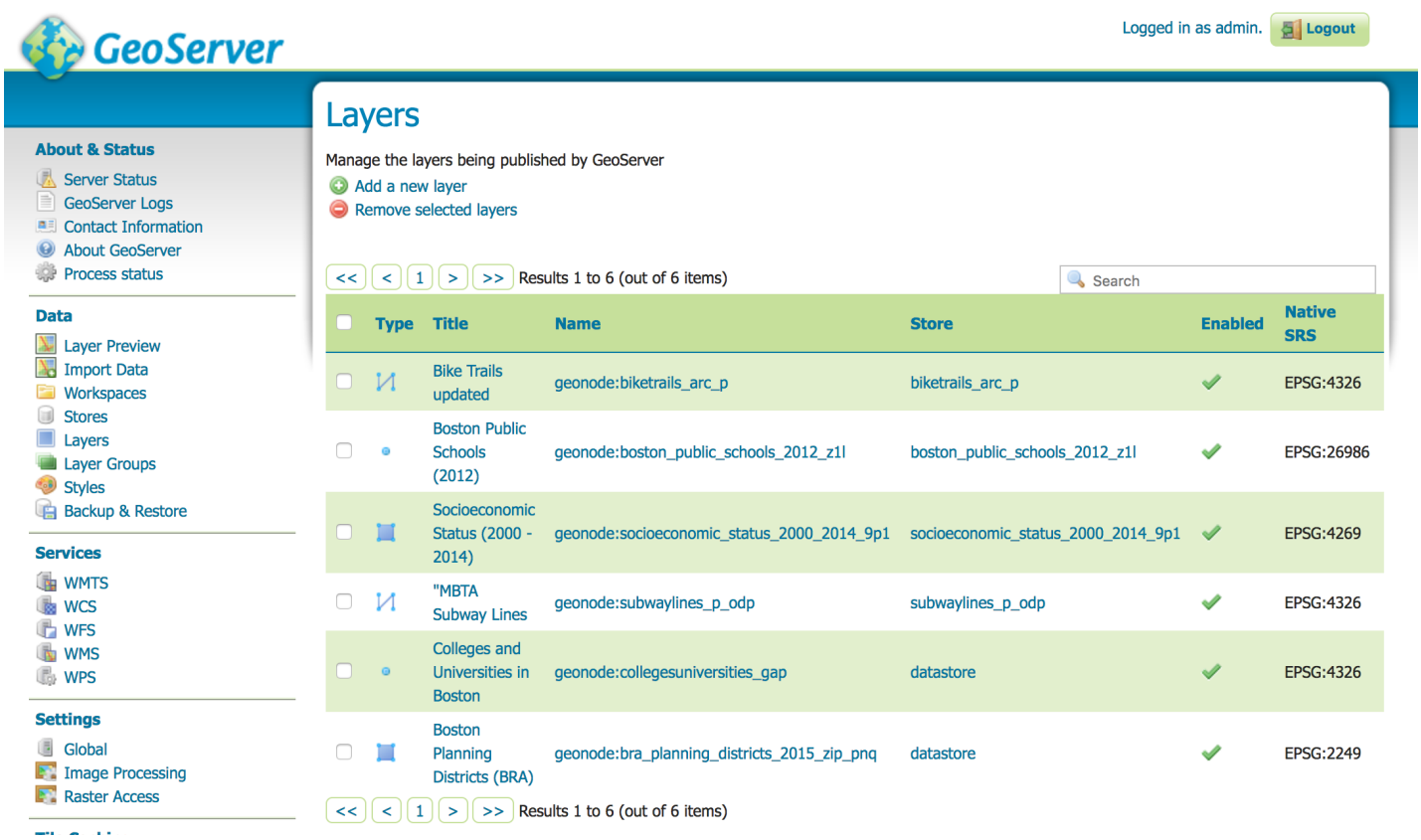
Now upload to GeoNode the following two shapefiles and fill the metadata as suggested below:

- first shapefile path: /workshop/data/shapefiles/CollegesUniversities_Gap.zip
  - Title: “Colleges and Universities in Boston”
  - Abstract: “Colleges and Universities in Boston. Point layer data collected by the Boston Redevelopment Authority.”
  - Regions: “United States of America”
  - Keywords: “boston, foss4g2017, education, society”
  - Category: “Society”
- second shapefile path: /workshop/data/shapefiles/bra_planning_districts_2015_zip_pnq.zip
  - Title: “Boston Planning Districts (BRA)”
  - Abstract: “Planning Districts in Boston, as determined by the Boston Redevelopment Authority (BRA).”
  - Regions: “United States of America”
  - Keywords: “boston, foss4g2017, boundaries, planning”
  - Category: “Planning Cadastre”

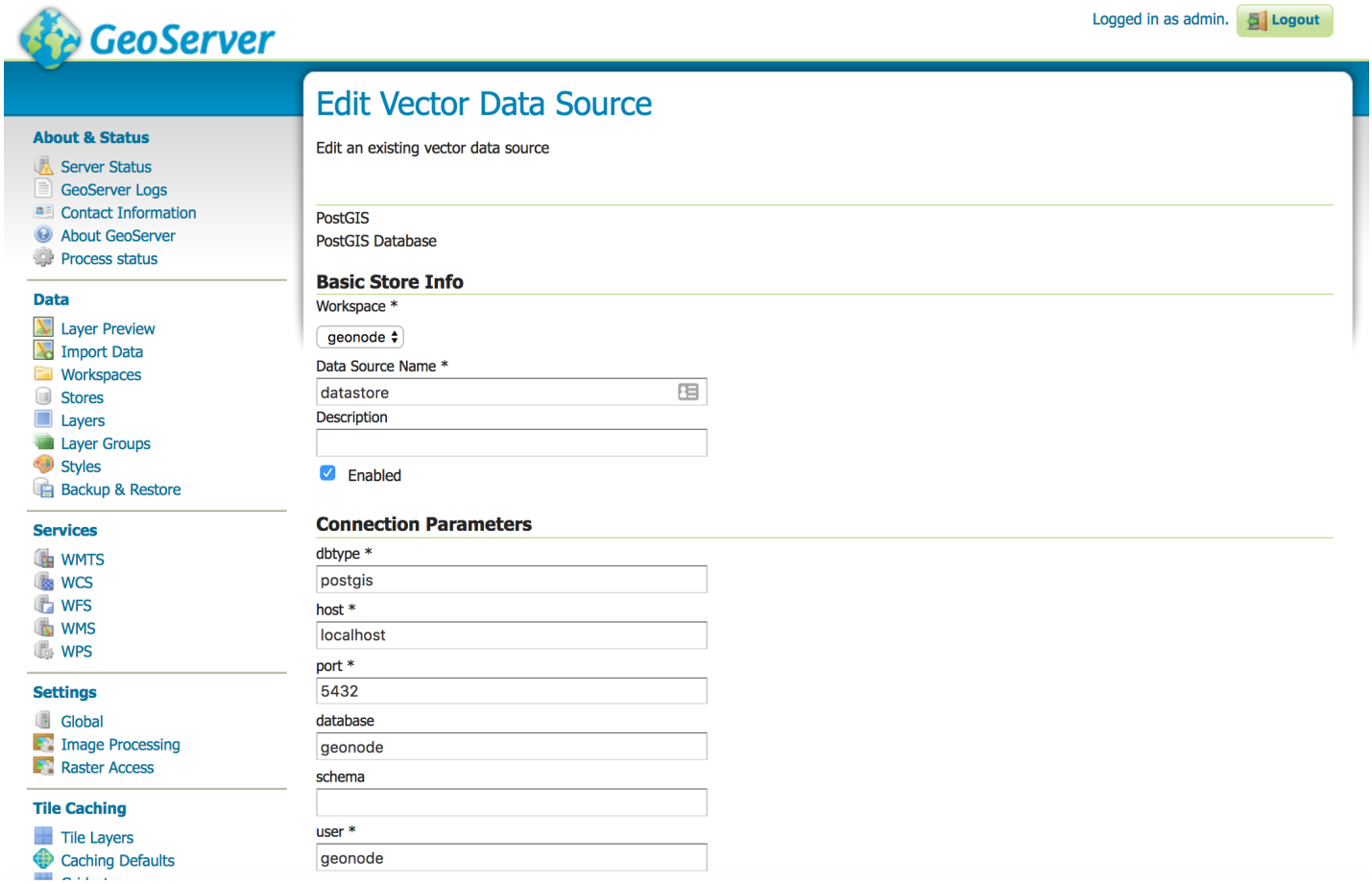
Check the new PostGIS store in GeoServer

You can easily check that the new layers are in a PostGIS store. Go to the GeoServer administrative site, and click on the *Layers* menu.

You will see that both the new layers (*geonode:collegesuniversities_gap* and *geonode:bra_planning_districts_2015_zip_pnq*) are in the same store, named *datastore*



If you click on the *datastore* link, you will get to the *datastore* store page. As you can see it is a PostGIS data store



A quick tour of PostGIS features

In this step of the tutorial you will have a quick tour of some of the most impressive PostGIS features. If you want to know more about it, two very useful resources are:

- Introduction to PostGIS workshop, by Boundless
- PostGIS documentation

Connect to the database and check which tables are there with the *\dt* command: the two layers you uploaded should be there, together with the *spatial_ref_sys*, a table needed by PostGIS which is created when you install PostGIS as an extension in the database

```
$ psql -h localhost -U geonode geonode
Password for user geonode:
psql (9.5.7)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, co
Type "help" for help.

geonode=# \dt
                      List of relations
 Schema |                Name                 | Type  |  Owner
--------+-------------------------------------+-------+----------
 public | bra_planning_districts_2015_zip_pnq | table | geonode
 public | collegesuniversities_gap            | table | geonode
 public | spatial_ref_sys                     | table | postgres
(3 rows)
```

When you install PostGIS in the database by using the *CREATE EXTENSION postgis* command you add a number of tables, views and functions on the PostgreSQL database.

The functions enrich the SQL and spatially enable the database. Let's read some of the features of the *bra_planning_districts_2015_zip_pnq* layer using a plain SQL **SELECT** statement:

```
geonode=# select * from collegesuniversities_gap limit 5;

 fid |                    the_geom                    | Status | Score | Match_ty
-----+------------------------------------------------+--------+-------+---------
   1 | 0101000020E6100000D0A9ED1E53C651C0E8E8080A992C4540 | M      |   100 | A
   2 | 0101000020E610000090F61DBFEBC451C0B88EA1250D2D4540 | M      |   100 | A
   3 | 0101000020E61000006C800F5D81C451C088C1760D452C4540 | M      |   100 | A
   4 | 0101000020E61000000476558EB8C551C06813BAFE632C4540 | M      |   100 | A
   5 | 0101000020E6100000E03957391EC651C04810C73CA42C4540 | M      |   100 | A
(5 rows)
```

Querying geometries

You can query PostGIS on the *the_geom* field to detect which is the geometry type of each feature using the **ST_GeometryType** function

```
geonode=# select ST_GeometryType(the_geom) from collegesuniversities_gap limit 5;
 st_geometrytype
-----------------
 ST_Point
 ST_Point
 ST_Point
 ST_Point
 ST_Point
(5 rows)
```

Try ST_GeometryType on the *bra_planning_districts_2015_zip_pnq* layer:

```
# select ST_GeometryType(the_geom) from bra_planning_districts_2015_zip_pnq limit 5;
 st_geometrytype
-----------------
 ST_MultiPolygon
 ST_MultiPolygon
 ST_MultiPolygon
 ST_MultiPolygon
 ST_MultiPolygon
(5 rows)
```

Now something more fancy: let's get the spatial reference system identifiers for each feature using the **ST_SRID** function

```
geonode=# select ST_SRID(the_geom) from collegesuniversities_gap limit 5;
 st_srid
---------
    4326
    4326
    4326
    4326
    4326
(5 rows)
```

```
geonode=#  select ST_SRID(the_geom) from bra_planning_districts_2015_zip_pnq limit 5;
 st_srid
```

```
---------
    2249
    2249
    2249
    2249
    2249
(5 rows)
```

4326 stands for EPSG:4326, WGS 84, while 2249 stands for EPSG:2249, NAD83 / Massachusetts Mainland

Let's get the centroid and the number of points of each polygon representing a planning district using the **ST_Centroid** and the **ST_NPoints** functions

```
geonode=# SELECT ST_Centroid(the_geom), ST_NPoints(the_geom) from bra_planning_distri
                   st_centroid                    | st_npoints
--------------------------------------------------+------------
 0101000020C9080000F27C37839004284155A2E98890974641 |        658
 0101000020C9080000FF0544DB919E274179DE9B37569C4641 |        182
 0101000020C9080000C63CA4EF908527410E33827844894641 |        286
 0101000020C9080000D245B41FA7CC2741F3ED8854527F4641 |        318
 0101000020C908000063180CD49892741F3905AEEED7F4641 |         71
(5 rows)
```

The geometry is stored as a binary value: it is possible to get a more human friendly representation using one of the geometry outputs function. For example let's get the representation of the geometry of each college/university in the WKT, EWKT, GeoJSON, GML, KML formats:

```
geonode=# SELECT ST_AsText(the_geom), ST_AsEWKT(the_geom), ST_AsGeoJSON(the_geom), ST

                st_astext                |                        st_asewkt
-----------------------------------------+-----------------------------------------
 POINT(-71.0988232918673 42.348420385695)  | SRID=4326;POINT(-71.0988232918673 42.348
 POINT(-71.0768888275727 42.3519637144904) | SRID=4326;POINT(-71.0768888275727 42.351
 POINT(-71.0703957225211 42.3458573179115) | SRID=4326;POINT(-71.0703957225211 42.345
 POINT(-71.08938940378 42.3468016060422)   | SRID=4326;POINT(-71.08938940378 42.34680
 POINT(-71.0955947257821 42.3487621280661) | SRID=4326;POINT(-71.0955947257821 42.348
(5 rows)
```

Transforming data

PostGIS let you transform data from one spatial reference system to another with the **ST_Transform** function.

Here is how you can get the WKT geometry for each district's centroid in both the original EPSG:2249 SRS and the EPSG:4326 SRS

```
geonode=# SELECT ST_AsText(ST_Centroid(the_geom)), ST_AsText(ST_Transform(ST_Centroid
                st_astext                 |                 st_astext
------------------------------------------+-------------------------------------------
 POINT(787016.256282715 2961185.06962995) | POINT(-71.0157631183247 42.3726239465675)
 POINT(773960.928253353 2963628.43444425) | POINT(-71.0640300464323 42.3795217918033)
 POINT(770760.468049907 2953864.94147337) | POINT(-71.0760547192716 42.3527748902606)
 POINT(779859.561922247 2948772.66042876) | POINT(-71.0424938530946 42.3386725283588)
 POINT(771236.901368649 2949083.86213886) | POINT(-71.0743804204793 42.3396489642072)
(5 rows)
```

Editing Geometries

With PostGIS you can run SQL **INSERT** and **UPDATE** commands to add new features or edit existing ones. Here is how to edit the position of the "New England School of Photography" in the *collegesuniversities_gap* layer by using an UPDATE SQL command combined with the **ST_MakePoint** function

```
geonode=# SELECT ST_AsText(the_geom) FROM collegesuniversities_gap WHERE "COLLEGE" LI
                st_astext
-----------------------------------------
 POINT(-71.0965929732347 42.3489959517937)
(1 row)

geonode=# UPDATE collegesuniversities_gap
geonode=# SET the_geom = ST_SetSRID(ST_MakePoint(-71.0965, 42.3489), 4326)
geonode=# WHERE "COLLEGE" LIKE '%Photography';
geonode=# UPDATE 1

geonode=# SELECT ST_AsText(the_geom) FROM collegesuniversities_gap WHERE "COLLEGE" LI
        st_astext
------------------------
 POINT(-71.0965 42.3489)
(1 row)
```

Spatial Joins

One of the most powerful features of PostGIS is the possibility to spatially joins one or more layers. For example you could get a list with colleges and universities for each planning district using the **ST_Contains** function.

As the two layers are using different spatial reference system, you need to transform one of the layer's geometry to the other spatial reference system

```
geonode=# SELECT d."PD" as district, c."COLLEGE" as college
geonode-# FROM bra_planning_districts_2015_zip_pnq as d, collegesuniversities_gap as
geonode-# WHERE ST_Contains(ST_Transform(d.the_geom, 4326), c.the_geom);
       district       |                      college
----------------------+-----------------------------------------------------
 Charlestown          | Bunker Hill Community College
 Charlestown          | MGH Institute Of Health Professions
 Back Bay/Beacon Hill | Bay State College
 Back Bay/Beacon Hill | Boston Architectural Center
 Back Bay/Beacon Hill | Butera School Of Art
 Back Bay/Beacon Hill | Fisher College
 Back Bay/Beacon Hill | Gibbs College
 Back Bay/Beacon Hill | Learning Institute For Beauty Sciences
 Back Bay/Beacon Hill | New England College of Optometry
 Back Bay/Beacon Hill | Simmons College
 South End            | Benjamin Franklin Institute of Technology
 Central              | Emerson College
 Central              | Massachusetts General Hospital Dietetic Internship
 Central              | New England College of Finance
 Central              | New England School Of Law
 Central              | North Bennet Street School
 Central              | Northeastern University
 Central              | Suffolk University
 Central              | Tufts University
 Central              | Urban College of Boston
 Fenway/Kenmore       | Art Institute of Boston at Lesley University
 Fenway/Kenmore       | Berklee College of Music
 Fenway/Kenmore       | Blaine The Beauty Career School-Boston
 Fenway/Kenmore       | Boston Conservatory
 Fenway/Kenmore       | Boston University
 Fenway/Kenmore       | Emmanuel College
 Fenway/Kenmore       | Harvard University
 Fenway/Kenmore       | Harvard University
 Fenway/Kenmore       | Massachusetts College of Art
 Fenway/Kenmore       | Massachusetts College of Pharmacy & Health Science
 Fenway/Kenmore       | New England Conservatory of Music
 Fenway/Kenmore       | Northeastern University
 Fenway/Kenmore       | School of the Museum of Fine Arts
 Fenway/Kenmore       | Simmons College
 Fenway/Kenmore       | Simmons College
 Fenway/Kenmore       | Wentworth Institute of Technology
 Fenway/Kenmore       | New England School Of Photography
```

```
 Allston/Brighton     | Harvard University
 Allston/Brighton     | Rets Technical Center
 North Dorchester     | University of Massachusetts Boston
 South Dorchester     | Laboure College
 West Roxbury         | Massachusetts School Of Professional Psychology
 Hyde Park            | Boston Baptist College
(43 rows)
```

Additional resources

- PostgreSQL Tutorial
- Introduction to PostGIS Workshop by Boundless
- The official PostgreSQL documentation
- The official PostGIS documentation
- PostGIS functions index

foss4g_2017_geonode_solr

Using GeoNode OGC services with external clients

In this tutorial you will see the GeoNode WMS and WFS OGC services in action with some different clients: QGIS (a desktop client), OpenLayers (a javascript client) and GDAL/OGR (a command line client).

Bonus step: try this kind of integration with other client such as gvSIG, OpenJUMP GIS, Esri ArcGIS, Leaflet etc...

QGIS

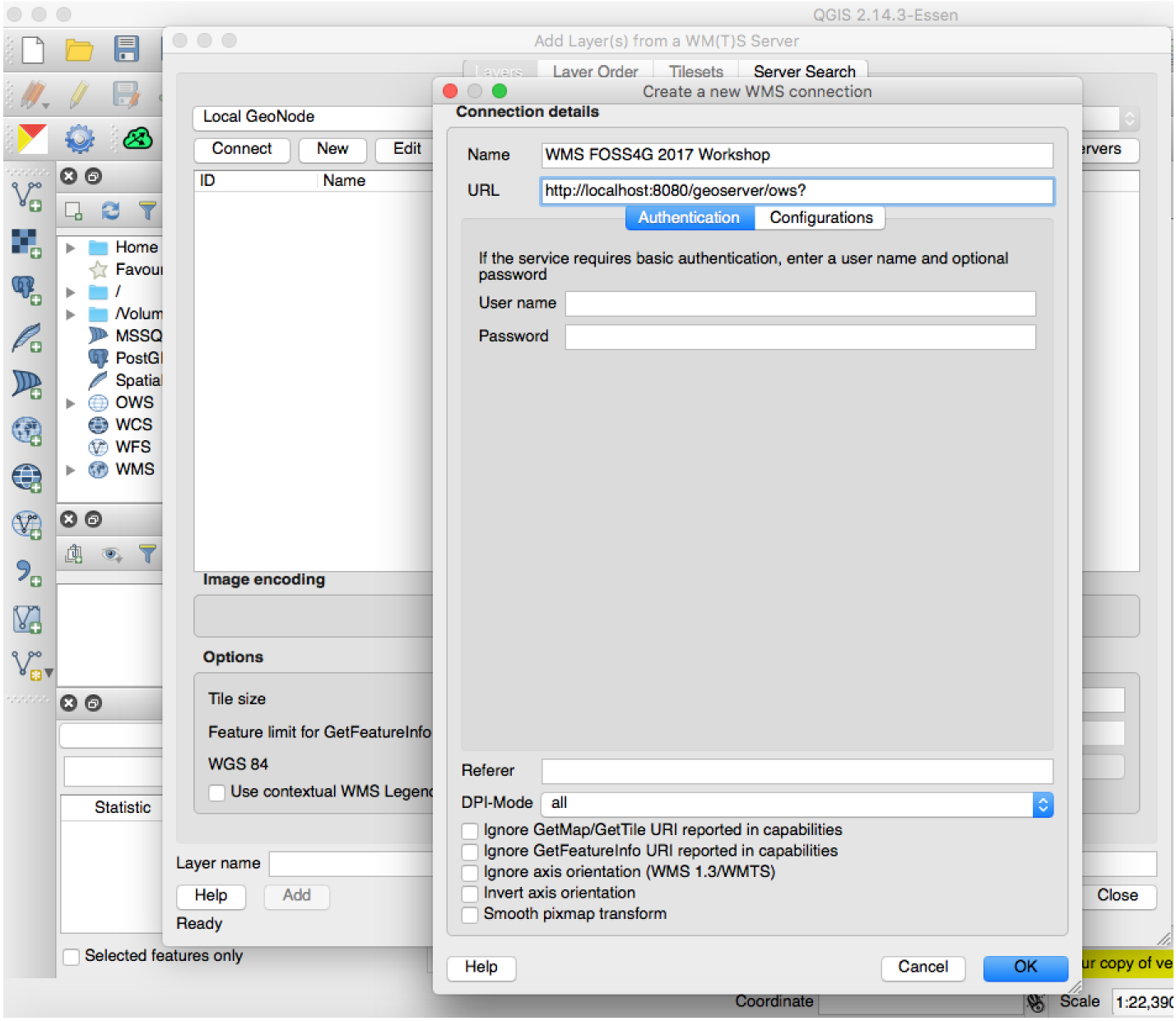
> QGIS (previously known as Quantum GIS) is a cross-platform free and open-source desktop geographic information system (GIS) application that supports viewing, editing, and analysis of geospatial data.

You used QGIS previously to connect to pycsw, the GeoNode's CSW catalogue. Here you will use QGIS to access the WMS and WFS GeoNode services.

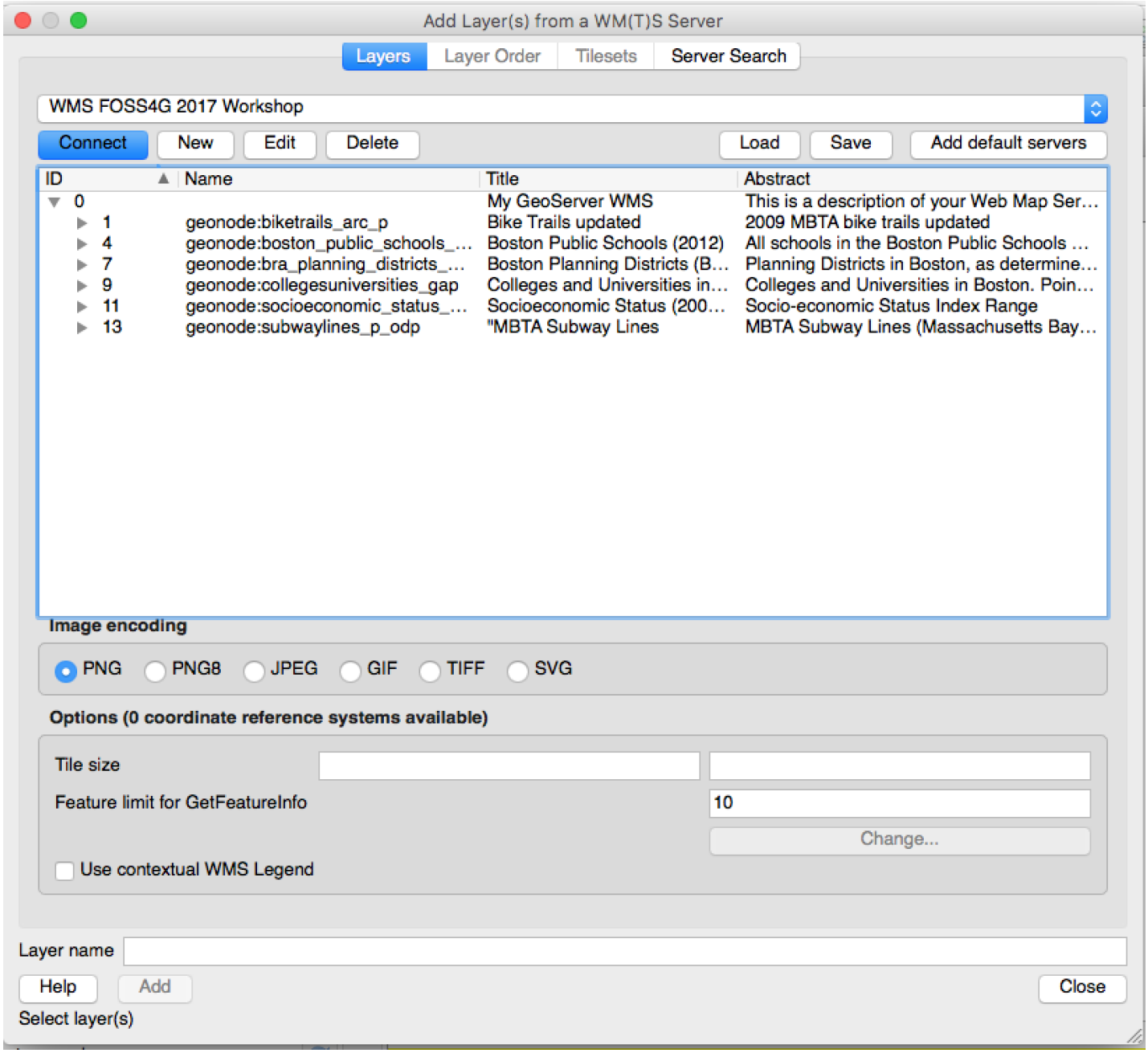
WMS

As a first thing, open the WMS connection dialog by clicking on the *Layer* menu, then *Add Layer* > *Add WMS/WMTS Layer* . From there, create a new WMS connection with these parameters:

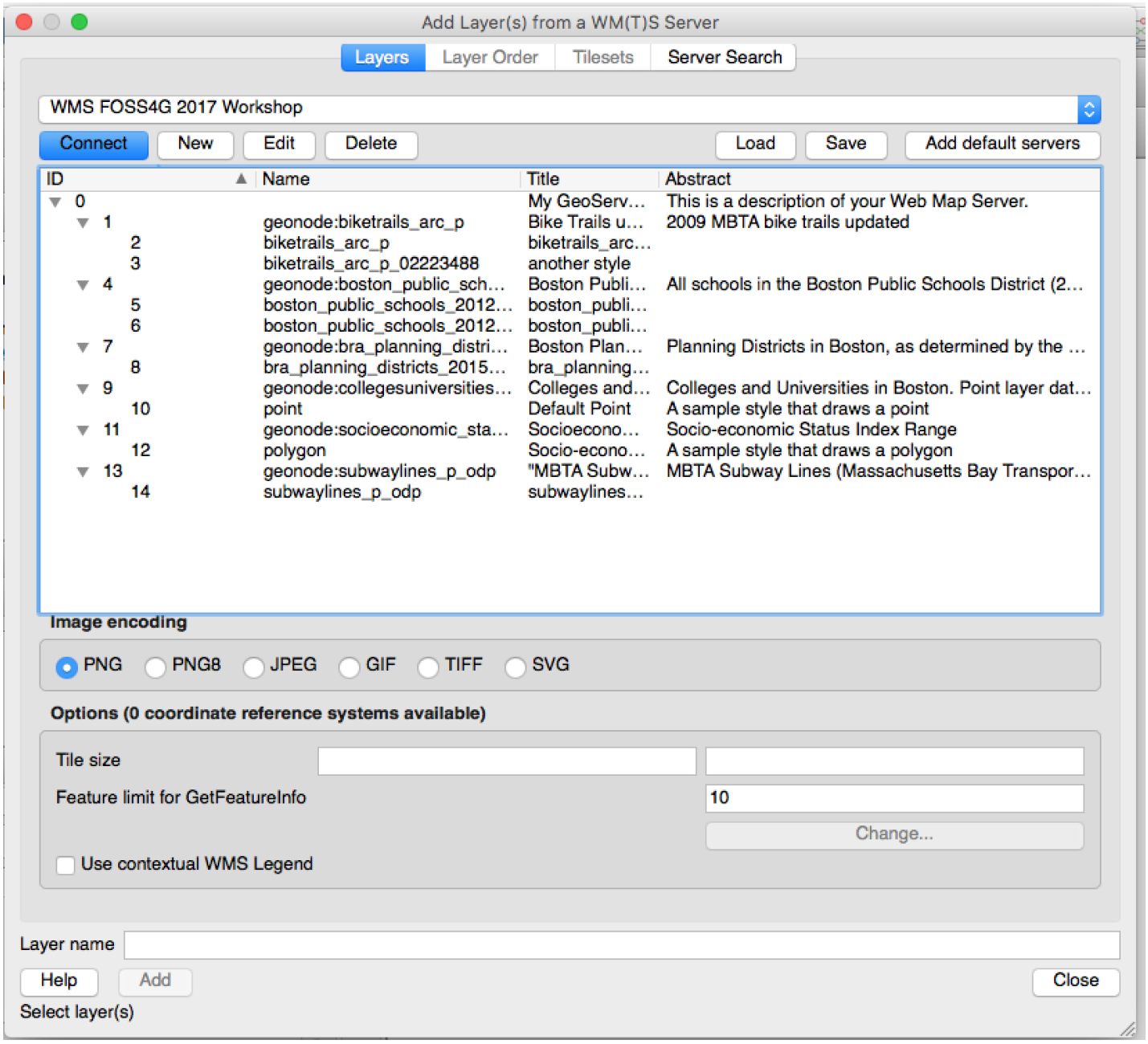
- Name: WMS FOSS4G 2017 Workshop
- url: http://localhost:8080/geoserver/ows?



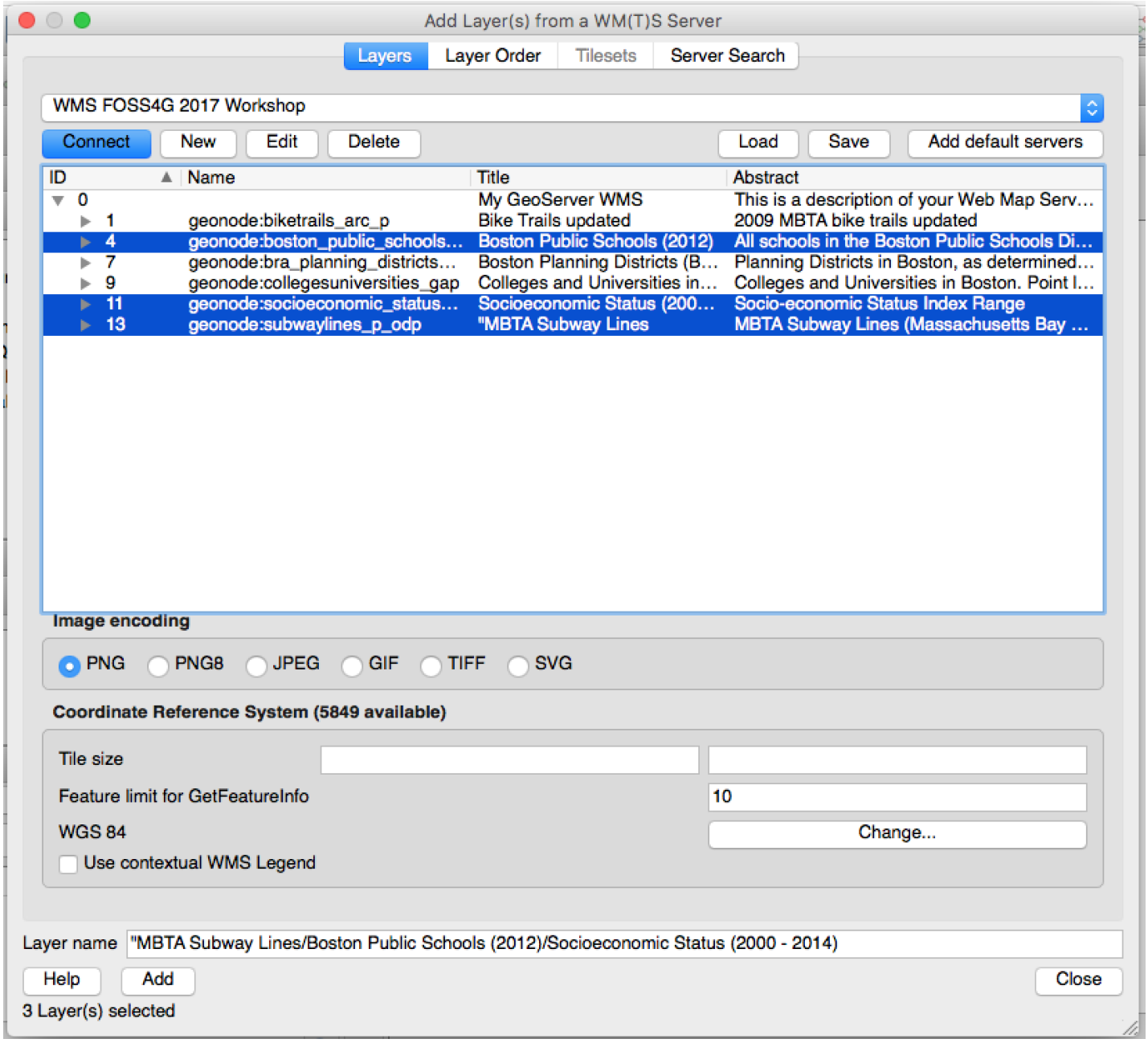
Now click on the *Connect* button, the list of layers should appear



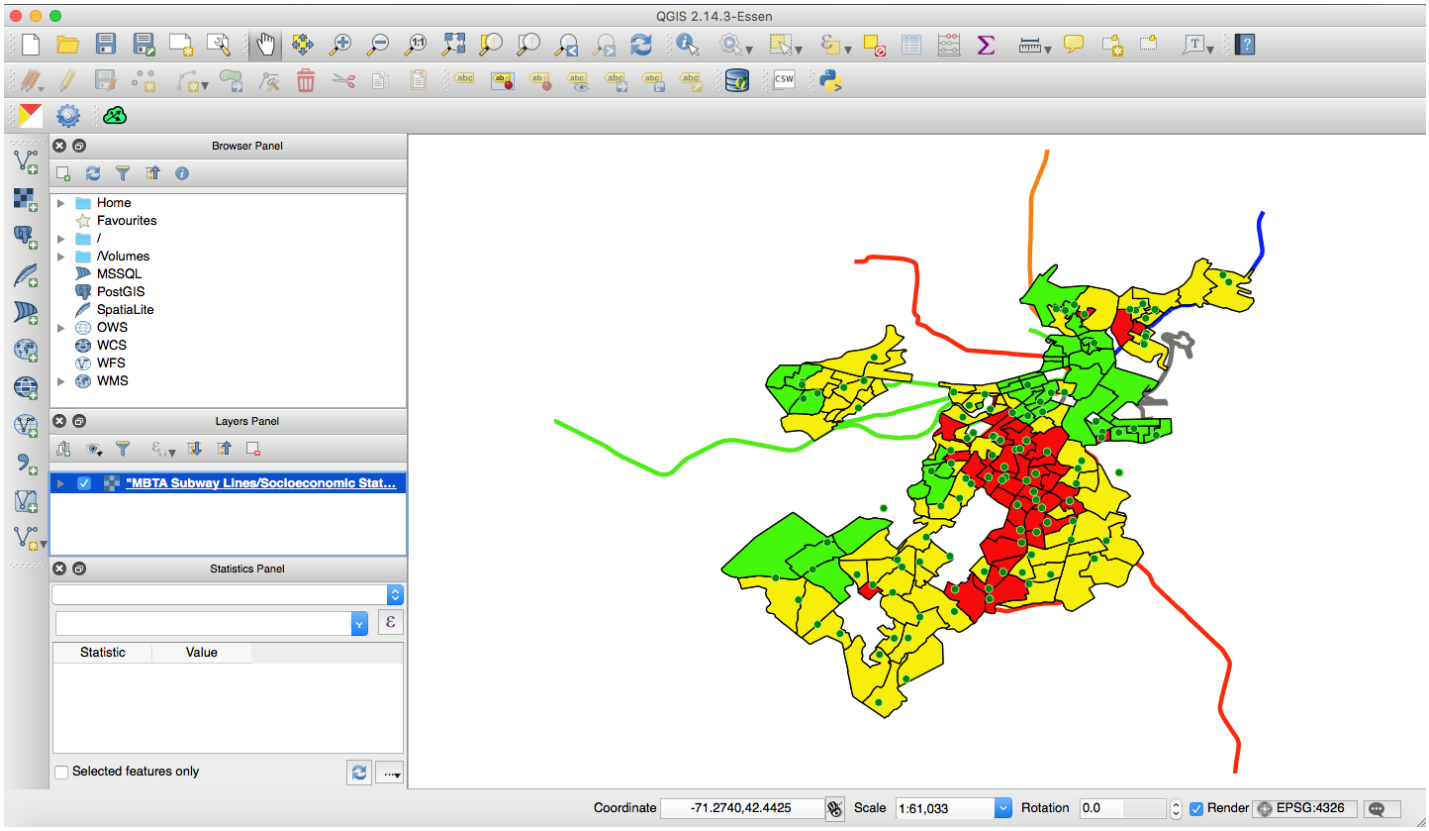
Note how each layer can be used in QGIS with each one of its styles



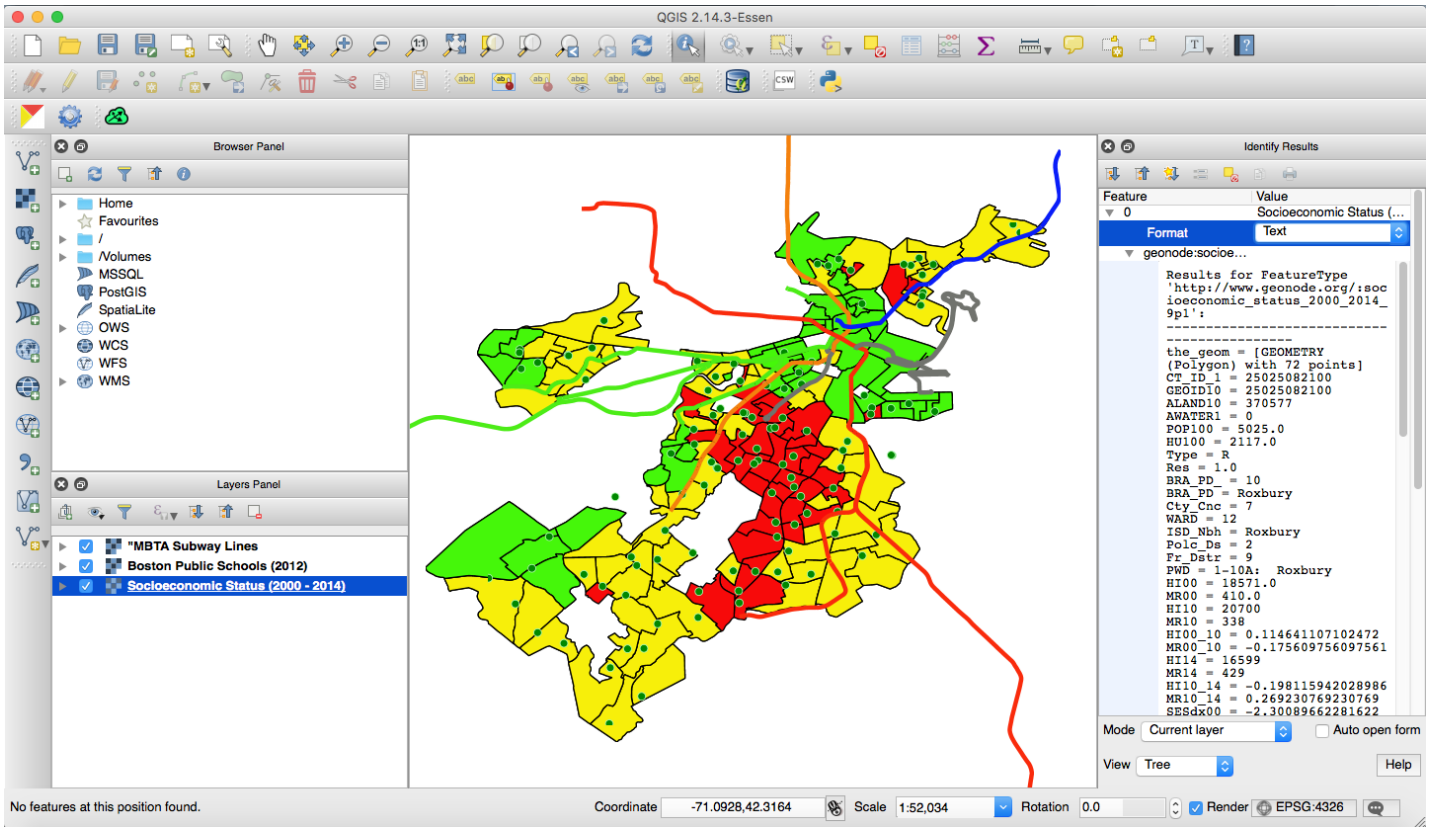
Select one or more layers and click on the *Add* button



Layers are added in the map as a layers group



Remove the layers group from the map and add each single layer to the map. Then browse the map and try to identify some of the features of each layer

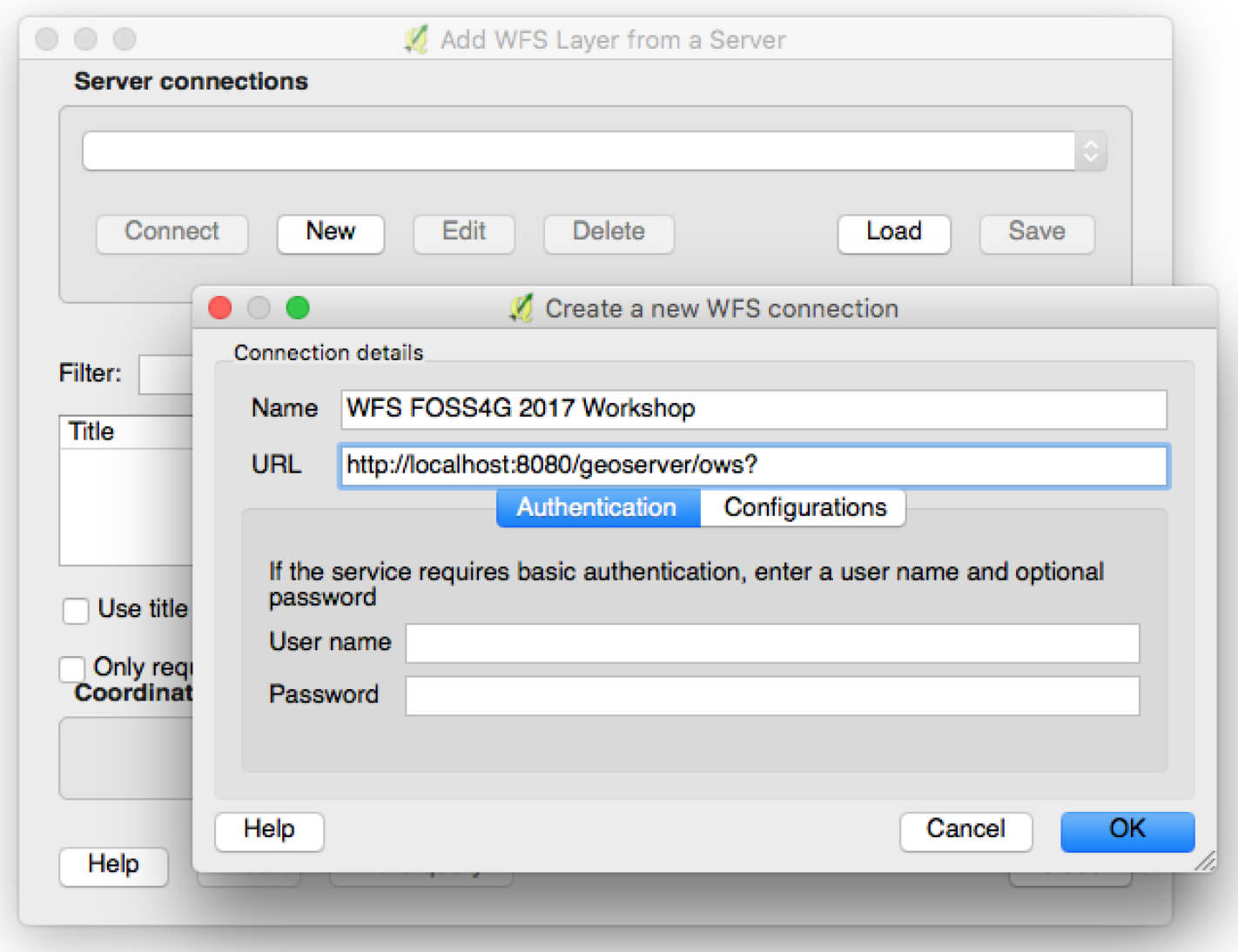


WFS

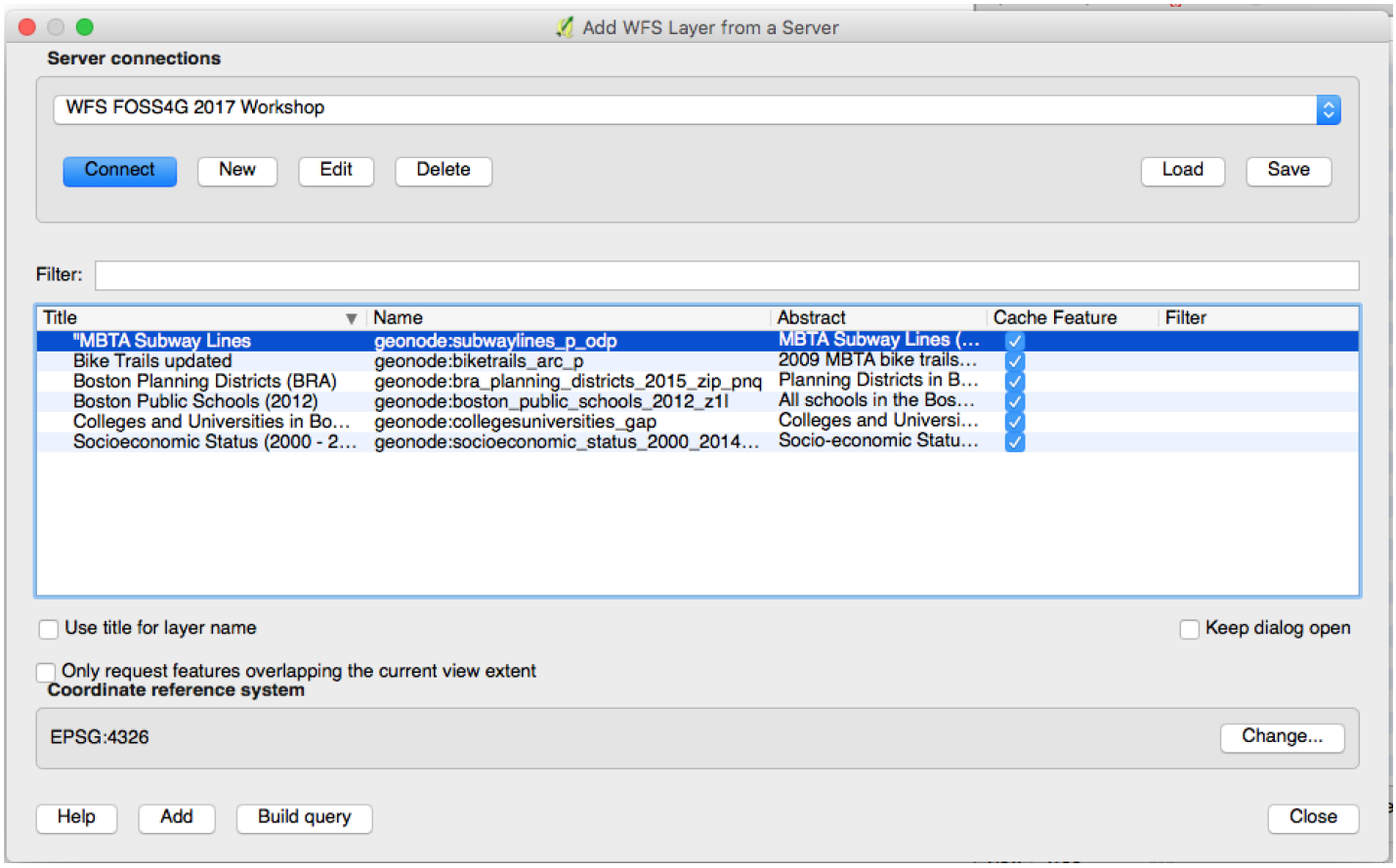
Open the WFS connection dialog by clicking on the *Layer* menu, then *Add Layer* > *Add WFS Layer*

Create new WFS connection using these parameters:

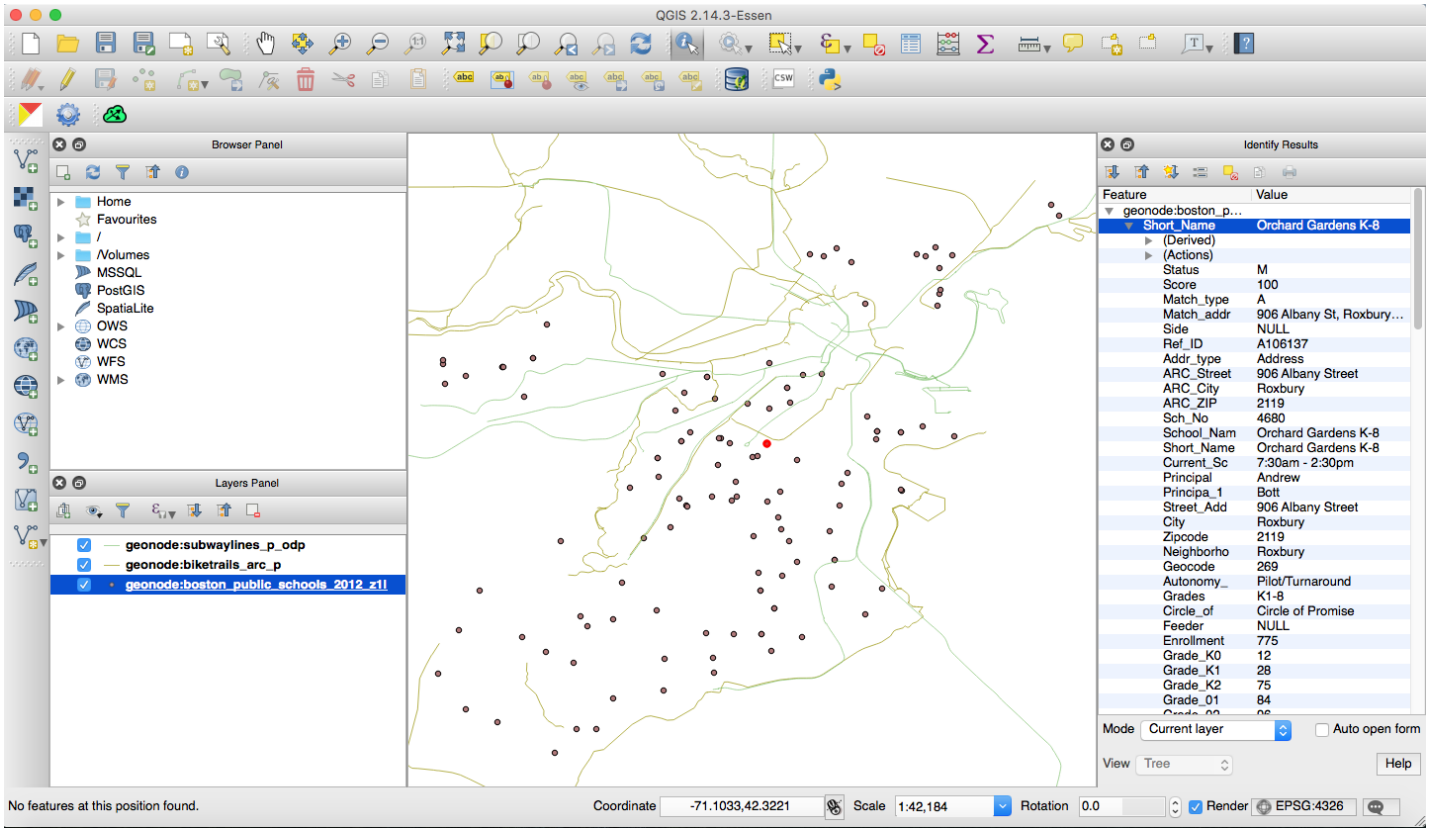
- Name: WFS FOSS4G 2017 Workshop
- url: http://localhost:8080/geoserver/ows?



Now click on the *Connect* button, the list of layers should appear



Select one or more layers and click on the *Add* button. Layers are added in the map. Then browse the map and try to identify some of the features of each layer



Layers have been added in QGIS as vector layers. Therefore it is possible to use most of the QGIS tools which are for vectors, for example try to:

- open the attribute table of a layer
- change the style of a layer
- query a layer to select some of its features
- export a layer to shapefile
- project a layer to a different spatial reference system
- edit a layer: this will use WFS-T behind the scenes and will work only if you use the credentials of a GeoServer user in the connect dialog. For example you can try with the *admin* GeoServer user (default password is "geoserver")

OpenLayers

> OpenLayers is an open source (provided under the 2-clause BSD License) JavaScript library for displaying map data in web browsers as slippy maps. It provides an API for building rich web-based geographic applications similar to Google Maps and Bing Maps.

OpenLayers is used by GeoNode itself, together with GeoExt, both in the layer details page and in the default map composer.

You can use OpenLayers to create custom viewers for your online mapping needs. Here you will see how to add some of the GeoNode layers to an OpenLayers map using the WMS endpoint.

For this purpose, you need an HTTP server to display an OpenLayers map in a html page. Install nginx (alternatively you could install httpd):

```
sudo apt-get install nginx
```

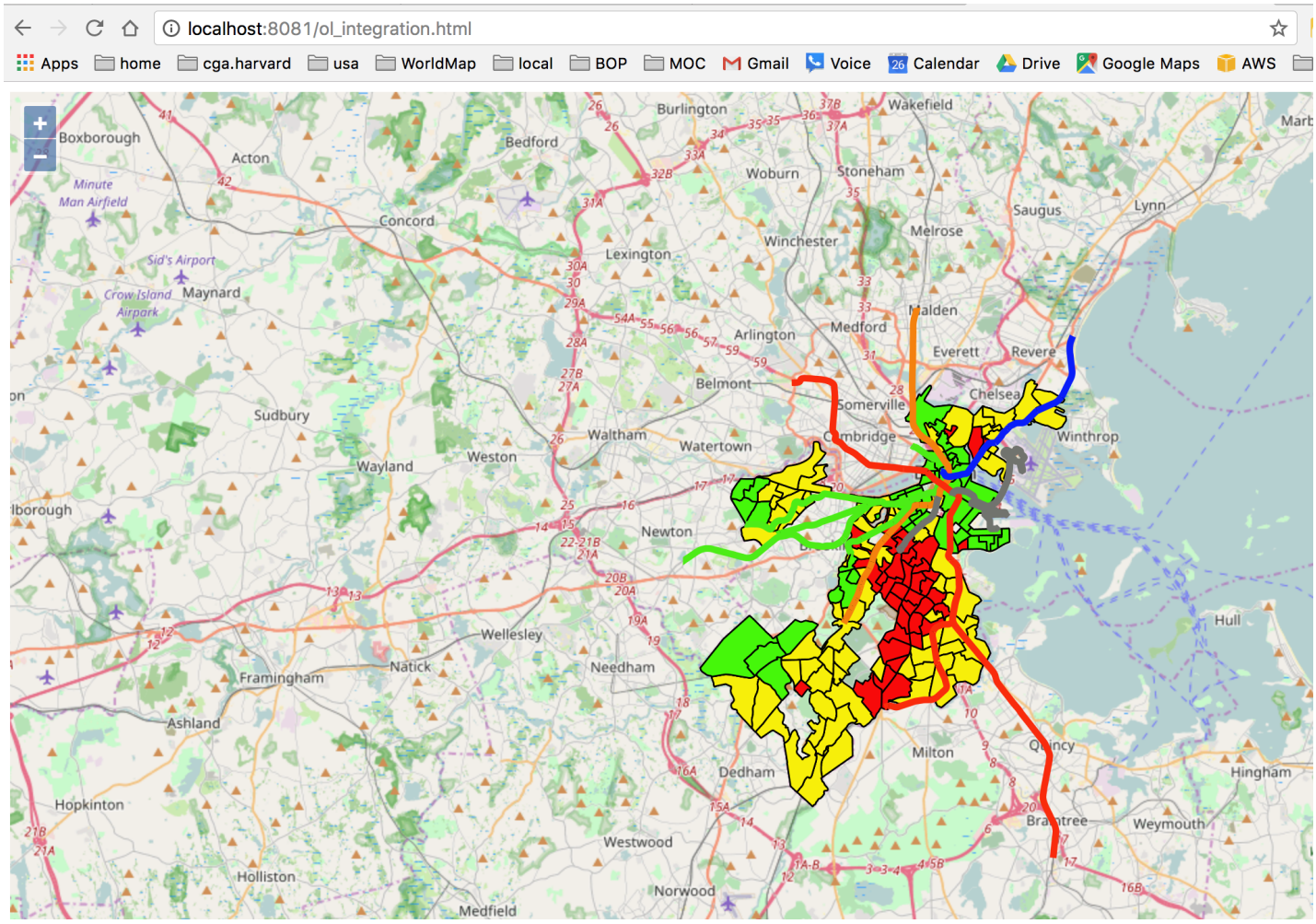
Check if nginx is running at: http://localhost:8081/ (we forwarded the guest 80 port to the host 8081 port in Vagrantfile).

Now create this html file at: /var/www/html/ol3_integration.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>OL WMS Integration</title>
    <link rel="stylesheet" href="https://openlayers.org/en/v4.2.0/css/ol.css" type="t
    <!-- The line below is only needed for old environments like Internet Explorer an
    <script src="https://cdn.polyfill.io/v2/polyfill.min.js?features=requestAnimation
```

```
    <script src="https://openlayers.org/en/v4.2.0/build/ol.js"></script>
  </head>
  <body>
    <div id="map" class="map"></div>
    <script>
      var layers = [
        new ol.layer.Tile({
          source: new ol.source.OSM()
        }),
        new ol.layer.Tile({
          extent: [-7926000, 5190846, -7895123, 5225000],
          source: new ol.source.TileWMS({
            url: 'http://localhost:8080/geoserver/wms',
            params: {
                      'LAYERS': 'geonode:socioeconomic_status_2000_2014_9p1,geonode:s
                      'TILED': true
                    },
            serverType: 'geoserver'
          })
        })
      ];
      var map = new ol.Map({
        layers: layers,
        target: 'map',
        view: new ol.View({
          center: [-7910000, 5210000],
          zoom: 11
        })
      });
    </script>
  </body>
</html>
```

Browse to http://localhost:8081/ol_integration.html and you should see an interactive OpenLayers map with two of the GeoNode layers (*socioeconomic_status_2000_2014_9p1* and *subwaylines_p_odp*)



GDAL/OGR

OGR, the vector part library of GDAL, can connect to WFS and WMS OGC services with the WFS driver and the WMS driver. Here you will have a look at the former one.

As for any geospatial format, GDAL/OGR can access WFS data programmatically (using several languages including Python) or using powerful command line tools. Here you will use the ogrinfo and ogr2ogr command line tools.

As a first thing, try to get the list of layers in GeoNode with ogrinfo:

```
$ ogrinfo -ro WFS:http://localhost:8080/geoserver/ows
INFO: Open of `WFS:http://localhost:8080/geoserver/ows'
      using driver `WFS' successful.
1: geonode:biketrails_arc_p (Multi Line String)
2: geonode:bra_planning_districts_2015_zip_pnq (Multi Polygon)
3: geonode:boston_public_schools_2012_z1l (Point)
4: geonode:collegesuniversities_gap (Point)
5: geonode:subwaylines_p_odp (Multi Line String)
6: geonode:socioeconomic_status_2000_2014_9p1 (Multi Polygon)
```

Now list the features of the *boston_public_schools_2012_z1l*, with a given filter (in this case a given zip code):

```
$ ogrinfo -ro WFS:http://localhost:8080/geoserver/ows geonode:boston_public_schools_2
INFO: Open of `WFS:http://localhost:8080/geoserver/ows'
      using driver `WFS' successful.
Layer name: geonode:boston_public_schools_2012_z1l
Geometry: Point
Feature Count: 6
Extent: (232573.480140, 890829.010078) - (234844.560065, 892465.220076)
...
```

Finally, using ogr2ogr, here is how you can export the features of the OGC WFS layer to a shapefile:

```
$ ogr2ogr /workshop/test.shp WFS:http://localhost:8080/geoserver/ows geonode:boston_p
```

Check if the shapefile was correctly generated at */workshop/test.shp*

foss4g_2017_geonode_solr

Using GeoNode management commands

GeoNode comes with some useful management commands that can be run by the GeoNode administrators:

- *importlayers* Imports a file or folder with geospatial files to GeoNode
- *updatelayers* Update the GeoNode application with data from GeoServer
- *fixsitename* Uses SITENAME and SITEURL to set the values of the default site object
- *delete_orphaned_files* Deletes orphaned files of deleted documents
- *delete_orphaned_thubms* Deletes orphaned thumbnails of deleted GeoNode resources (Layers, Maps and Documents)
- *fix_baselayers* Fix base layers for all of the GeoNode maps or for a given map

In this tutorial you will become familiar with *updatelayers* and *importlayers*

updatelayers

updatelayers update the GeoNode layers with data from GeoServer. This is useful to add data in formats that are not supported in GeoNode by default, like ArcSDE, Oracle Spatial, ECW etc...

The updatelayers command provides several options that can be used to control how layer information is read from GeoServer and updated in GeoNode. You can have a look at all the command's options by running it. Open the Django shell and run the command with the –help option to see all of the available options:

```
$ python manage.py updatelayers --help
Usage: manage.py updatelayers [options]

Update the GeoNode application with data from GeoServer

Options:
  --version             show program version number and exit
  -h, --help            show this help message and exit
  -v VERBOSITY, --verbosity=VERBOSITY
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
```

```
    --settings=SETTINGS   The Python path to a settings module, e.g.
                          "myproject.settings.main". If this is not provided, the
                          DJANGO_SETTINGS_MODULE environment variable will be
                          used.
    --pythonpath=PYTHONPATH
                          A directory to add to the Python path, e.g.
                          "/home/djangoprojects/myproject".
    --traceback           Raise on CommandError exceptions
    --no-color            Do not colorize the command output.
    -i, --ignore-errors   Stop after any errors are encountered.
    --skip-unadvertised   Skip processing unadvertised layers from GeoSever.
    --skip-geonode-registered
                          Just processing GeoServer layers still not registered
                          in GeoNode.
    --remove-deleted      Remove GeoNode layers that have been deleted from
                          GeoSever.
    -u USER, --user=USER  Name of the user account which should own the imported
                          layers
    -f FILTER, --filter=FILTER
                          Only update data the layers that match the given
                          filter
    -s STORE, --store=STORE
                          Only update data the layers for the given geoserver
                          store name
    -w WORKSPACE, --workspace=WORKSPACE
                          Only update data on specified workspace
    -p PERMISSIONS, --permissions=PERMISSIONS
                          Permissions to apply to each layer
```

In this tutorial you will create a new layer in GeoServer and then register it in GeoNode using the updatelayers command.

The layer you will create will be based on a PostGIS view. The PostGIS view is a MultiPolygon layer composed by a buffer area of 1000 meters for each point from the *collegesuniversities_gap* layer.

Connect to the *geonode* database in PostgreSQL with the *geonode* user

```
$ psql -h localhost -U geonode geonode
Password for user geonode:
psql (9.5.7)
Type "help" for help.

geonode=#
```

Create the PostGIS view. Note that it is necessary to transform the geometry of the original layer to a projected system in order to buffer the features using a distance

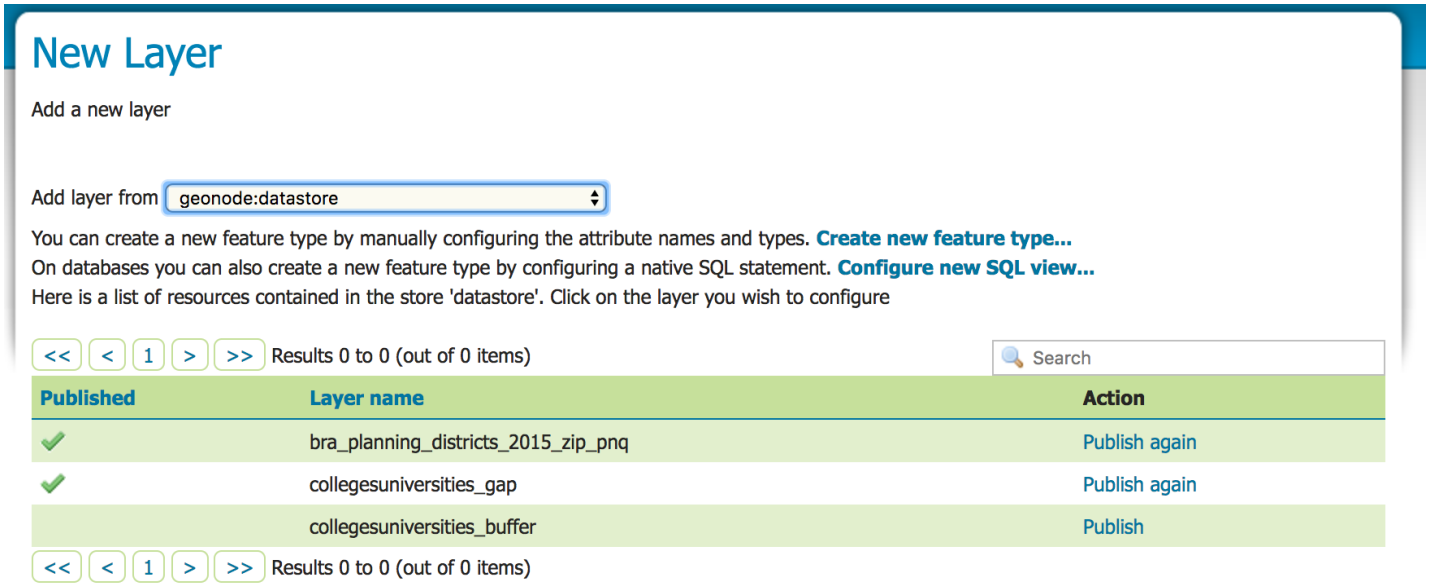
```
geonode=# CREATE TABLE collegesuniversities_buffer AS
geonode-# SELECT ST_Buffer(ST_Transform(the_geom, 2249), 1000) as the_geom, "ZIPCODE"
```

Now check if the view was correctly created and registered in PostGIS by checking the *geometry_columns* view

```
geonode=# select * from geometry_columns;

 f_table_catalog | f_table_schema |            f_table_name              | f_geometry_
-----------------+----------------+--------------------------------------+------------
 geonode         | public         | collegesuniversities_gap             | the_geom
 geonode         | public         | bra_planning_districts_2015_zip_pnq | the_geom
 geonode         | public         | collegesuniversities_buffer          | the_geom
(3 rows)
```

Time to add the view in GeoServer. Using the GeoServer administrative site, click on *Layers* and then on *Add a new layer*

Select *geonode:datastore* in the *Add layer from* store list



Then click on the *Publish* action for the *collegesuniversities_buffer* layer. In the *Edit Layer* dialog fill the *Native Bounding Box* text boxes by clicking the *Compute from data* link. Fill the *Lat/Lon Bounding Box* text boxes by clicking the *Compute from native bounds* link.

Bounding Boxes

Native Bounding Box

Min X	Min Y	Max X	Max Y
742,265.5663826	2,916,730.202410	779,862.7020479	2,966,168.144219

Compute from data
Compute from SRS bounds

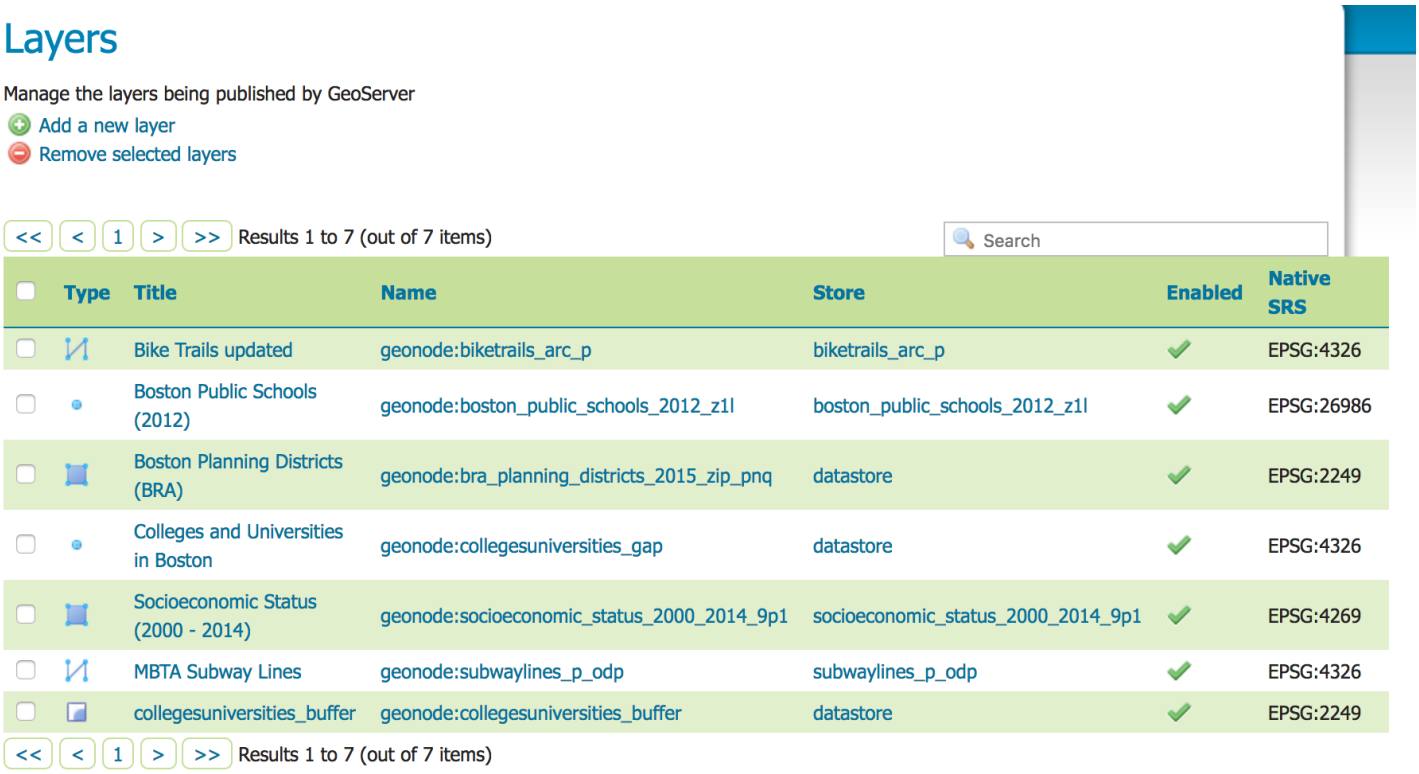
Lat/Lon Bounding Box

Min X	Min Y	Max X	Max Y
-71.18198404638	42.25074630483	-71.04213686217	42.38687554034

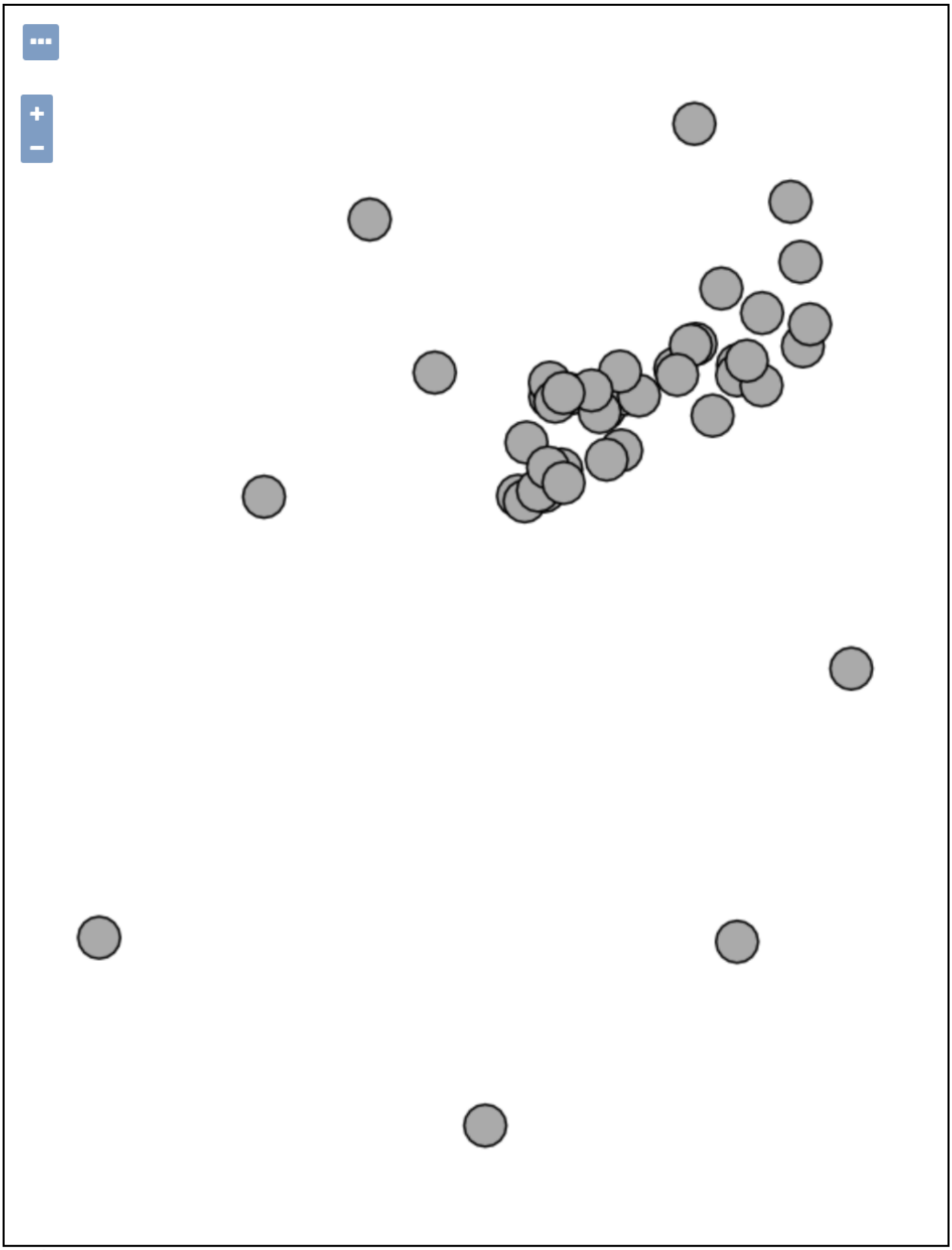
Compute from native bounds

Finally click on the *Save* button.

Now you should see a new layer, *geonode:collegesuniversities_buffer* in *Layers*



Check if the layer is correctly displayed in the *Layer Preview* section



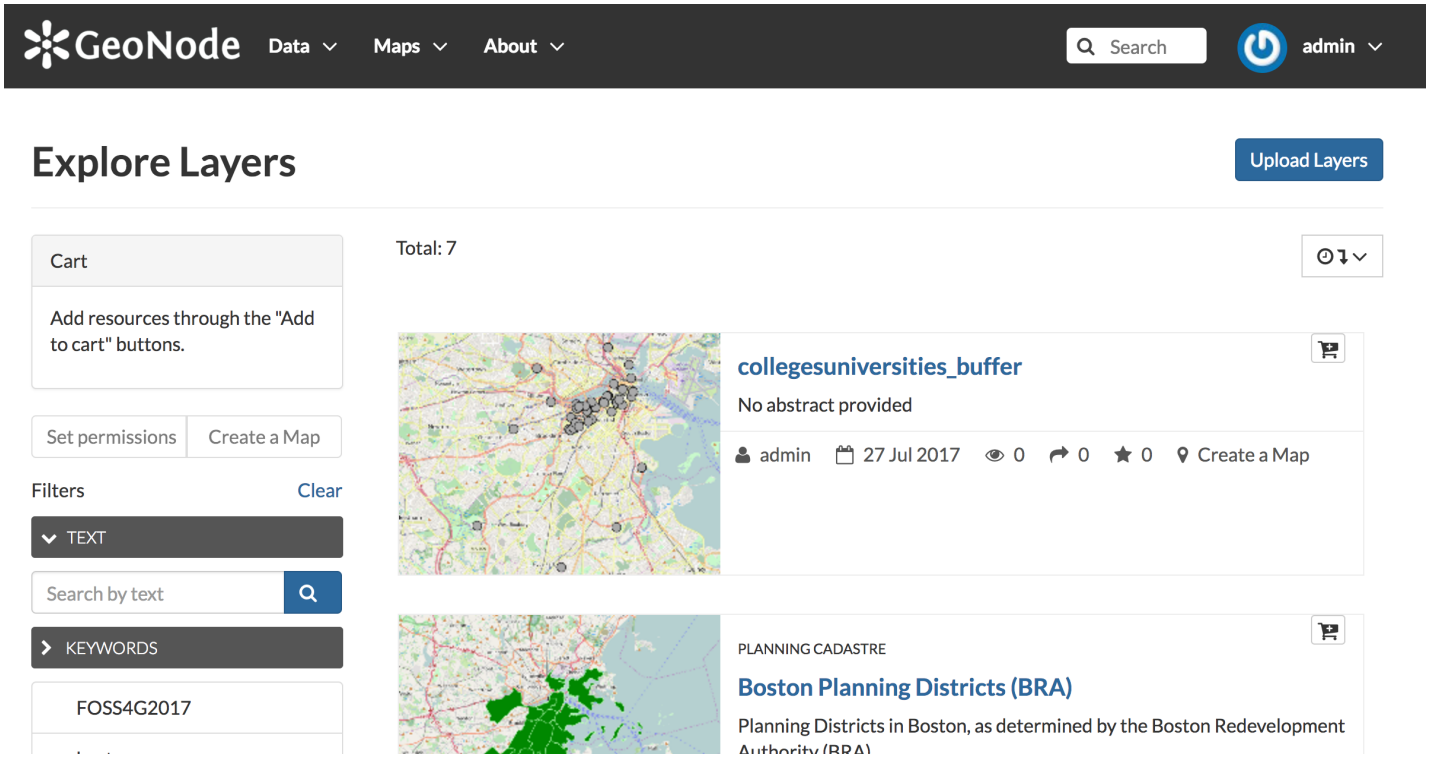
Scale = 1 : 273K

Click on the map to get feature info

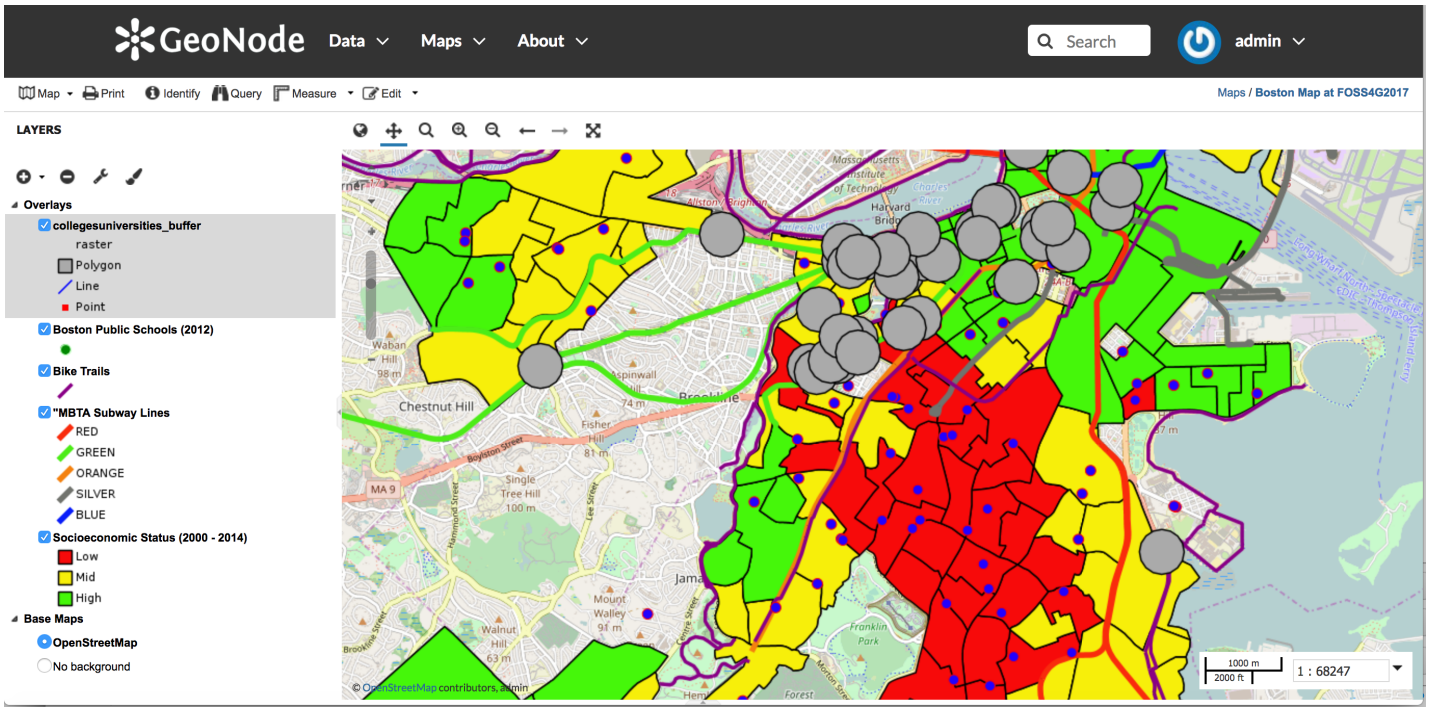
Now run the updatelayers command to register the GeoServer layer in GeoNode

```
$ python manage.py updatelayers --skip-geonode-registered
System check identified some issues:

WARNINGS:
base.ResourceBase.tkeywords: (fields.W340) null has no effect on ManyToManyField.
Inspecting the available layers in GeoServer ...
Found 1 layers, starting processing

[created] Layer collegesuniversities_buffer (1/1)

Finished processing 1 layers in 7.0 seconds.

1 Created layers
0 Updated layers
0 Failed layers
7.000000 seconds per layer
```

Check if the new layer is in the GeoNode layers page



Add the layer to the map you previously created



importlayers

The *importlayers* command imports a file or folder with geospatial files to GeoNode.

It supports data in Shapefile and GeoTIFF format. It also picks up the styles if a .sld file is present.

Check the importlayers options:

```
$ python manage.py importlayers --help
Usage: manage.py importlayers [options] path [path...]

Brings a data file or a directory full of data files into a GeoNode site.  Layers are

Options:
  --version             show program s version number and exit
  -h, --help            show this help message and exit
  -v VERBOSITY, --verbosity=VERBOSITY
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings=SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn t provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath=PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Raise on CommandError exceptions
  --no-color            Don t colorize the command output.
```

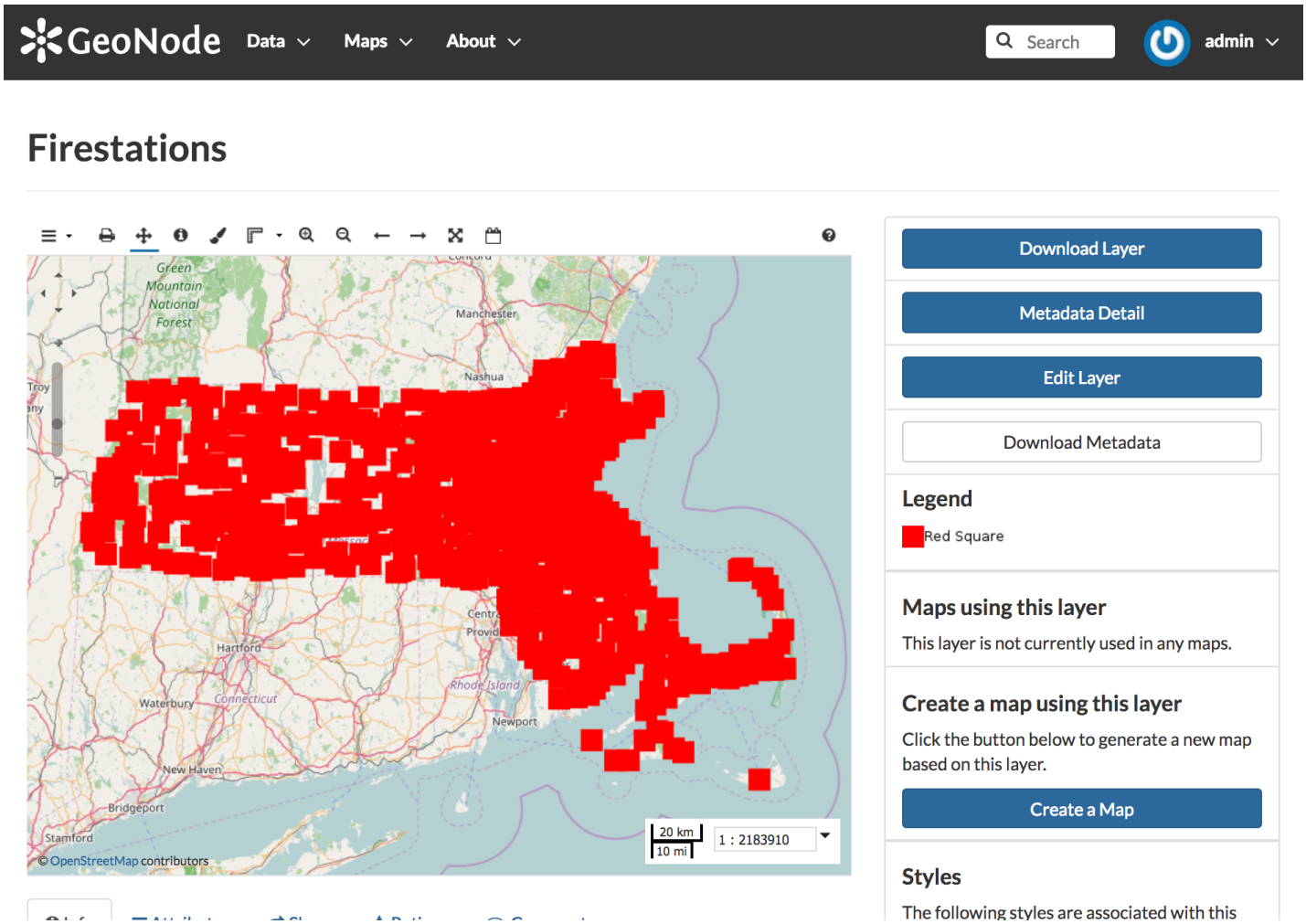
```
  -u USER, --user=USER  Name of the user account which should own the imported
                        layers
  -i, --ignore-errors   Stop after any errors are encountered.
  -o, --overwrite       Overwrite existing layers if discovered (defaults
                        False)
  -k KEYWORDS, --keywords=KEYWORDS
                        The default keywords, separated by comma, for the
                        imported layer(s). Will be the same for all imported
                        layers                    if multiple imports are
                        done in one command
  -c CATEGORY, --category=CATEGORY
                        The category for the                    imported
                        layer(s). Will be the same for all imported layers
                        if multiple imports are done in one command
  -r REGIONS, --regions=REGIONS
                        The default regions, separated by comma, for the
                        imported layer(s). Will be the same for all imported
                        layers                    if multiple imports are
                        done in one command
  -t TITLE, --title=TITLE
                        The title for the                     imported
                        layer(s). Will be the same for all imported layers
                        if multiple imports are done in one command
  -d DATE, --date=DATE  The date and time for the imported layer(s). Will be
                        the same for all imported layers if multiple imports
                        are done in one command. Use quotes to specify both
                        the date and time in the format 'YYYY-MM-DD HH:MM:SS'.
  -p, --private         Make layer viewable only to owner
  -m, --metadata_uploaded_preserve
                        Force metadata XML to be preserved
```

Now import the */workshop/data/shapefiles/FIRESTATIONS_PT_MEMA.zip* shapefile with importlayers. You will use the –title, –category and –keywords options to specify some of the layer's metadata

```
$ python manage.py importlayers --title Firestations --category Location --keywords b

System check identified some issues:
WARNINGS:
base.ResourceBase.tkeywords: (fields.W340) null has no effect on ManyToManyField.
[created] Layer for '/tmp/tmpGuXy83/FIRESTATIONS_PT_MEMA.shp' (1/1)

Finished processing 1 layers in 11.0 seconds.

1 Created layers
0 Updated layers
0 Skipped layers
```

```
0 Failed layers
11.000000 seconds per layer
```

Check if the layer is now available in GeoNode, and then add it to the map you created previously



foss4g_2017_geonode_solr

Using a search engine with GeoNode: a quick tour of Solr

> Solr is the popular, blazing fast open source enterprise search platform from the Apache Lucene project. Its major features include powerful full-text search, hit highlighting, faceted search, dynamic clustering, database integration, rich document (e.g., Word, PDF) handling, and geospatial search. Solr is highly scalable, providing distributed search and index replication, and it powers the search and navigation features of many of the world's largest internet sites.

> Solr is written in Java and runs as a standalone full-text search server within a servlet container such as Jetty. Solr uses the Lucene Java search library at its core for full-text indexing and search, and has REST-like HTTP/XML and JSON APIs that make it easy to use from virtually any programming language. Solr's powerful external configuration allows it to be tailored to almost any type of application without Java coding, and it has an extensive plugin architecture when more advanced customization is required.

As CMS are adding Solr (or Elasticsearch) to the stack to improve user search experience, Solr can be very useful in the context of an SDI as it provides features which are not traditionally provided by GeoPortals and OGC standards.

In this tutorial you will get an understanding about how to use Solr together with GeoNode to provide to end users a better search experience with features such as:

- full text metadata search
- scored results
- proximity matching text search
- boosts
- date ranges metadata search
- keyword, temporal and spatial faceting
- Solr spatial features

If you are willing to improve your understanding of Solr some very useful resources are:

- Solr quick start

- Apache Solr Reference Guide
- Apache Solr Enterprise Search Server by David Smiley

Create a core

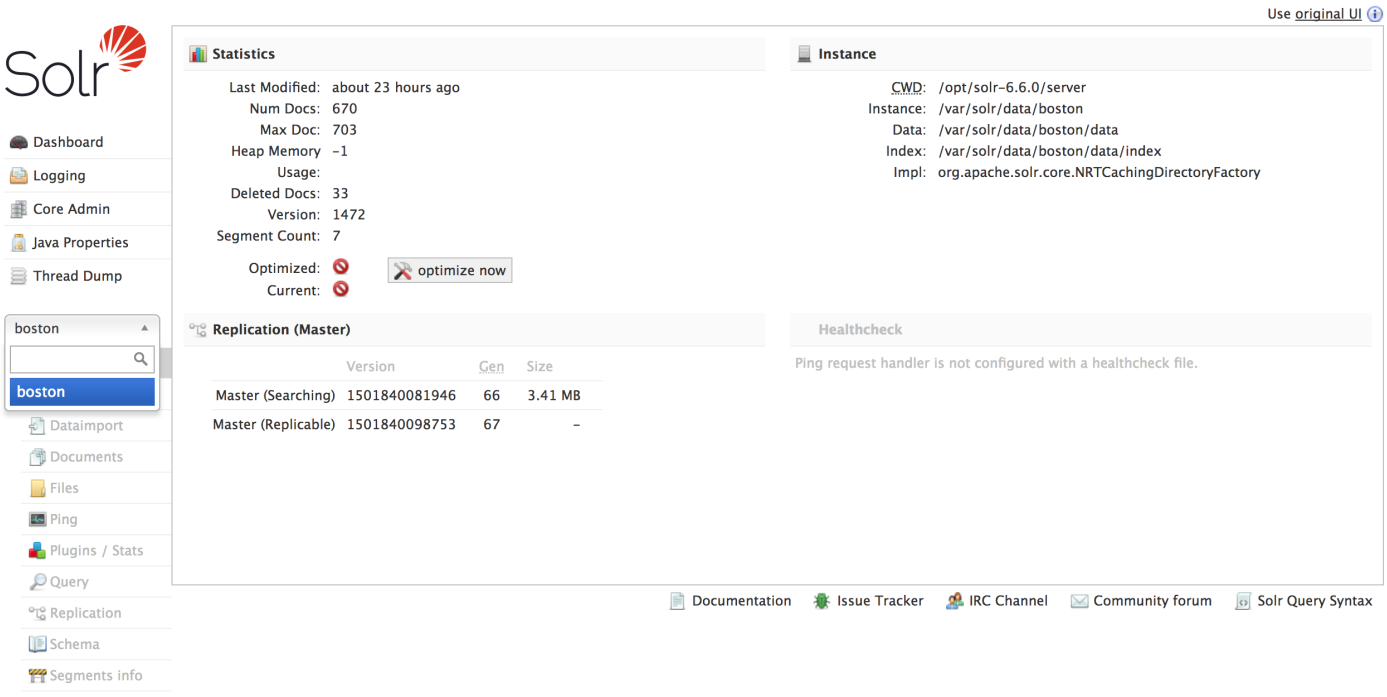
You already installed in a previous tutorial Solr, and it should be running at http://localhost:8983/solr

As a first thing create a core for the aims of this tutorial. You will name the core *boston*

```
$ sudo su solr
$ cd /opt/solr-6.6.0/bin/
$ ./solr create -c boston
$ exit
```

Now check if the new core is available from the Solr admin interface, which is running at: http://localhost:8983/solr/#/

You should be able to see *boston* in the *core selector* select list



In case you need to restart from scratch, here is how to remove the core:

```
$ sudo su solr
$ cd /opt/solr-6.6.0/bin/
$ ./solr delete -c boston
```

Create a schema

Solr stores details about the field types and fields it is expected to understand in a schema file. The name and location of this file may vary depending on how you initially configured Solr or if you modified it later.

managed-schema is the name for the schema file Solr uses by default to support making schema changes at runtime via the Schema API, or Schemaless Mode features.

You will create the schema for the *boston* core by sending POST JSON requests to the Solr API (http://localhost:8983/solr/boston/schema endpoint) with Python and the great Requests Python library

Create a directory where you will add the Python script, a **init**.py file into it and then the Python script itself (named *solr_schema*):

```
$ mkdir /workshop/foss4g_scripts
$ touch /workshop/foss4g_scripts/__init__.py
$ touch /workshop/foss4g_scripts/solr_schema.py
```

The **init**.py files is required to make Python treat the directory as containing packages. This way you will be able to import the solr_schema package in the Django shell.

Add the following Python code to the *solr_schema.py* script:

```
import requests


schema_url = "http://localhost:8983/solr/boston/schema"


def create_schema():
    """
    Create the schema in the solr core.
    """

    # create a special type to draw better heatmaps
    location_rpt_quad_5m_payload = {
        "add-field-type": {
            "name": "location_rpt_quad_5m",
            "class": "solr.SpatialRecursivePrefixTreeFieldType",
            "geo": False,
            "worldBounds": "ENVELOPE(-180, 180, 180, -180)",
            "prefixTree": "packedQuad",
            "distErrPct": "0.025",
```

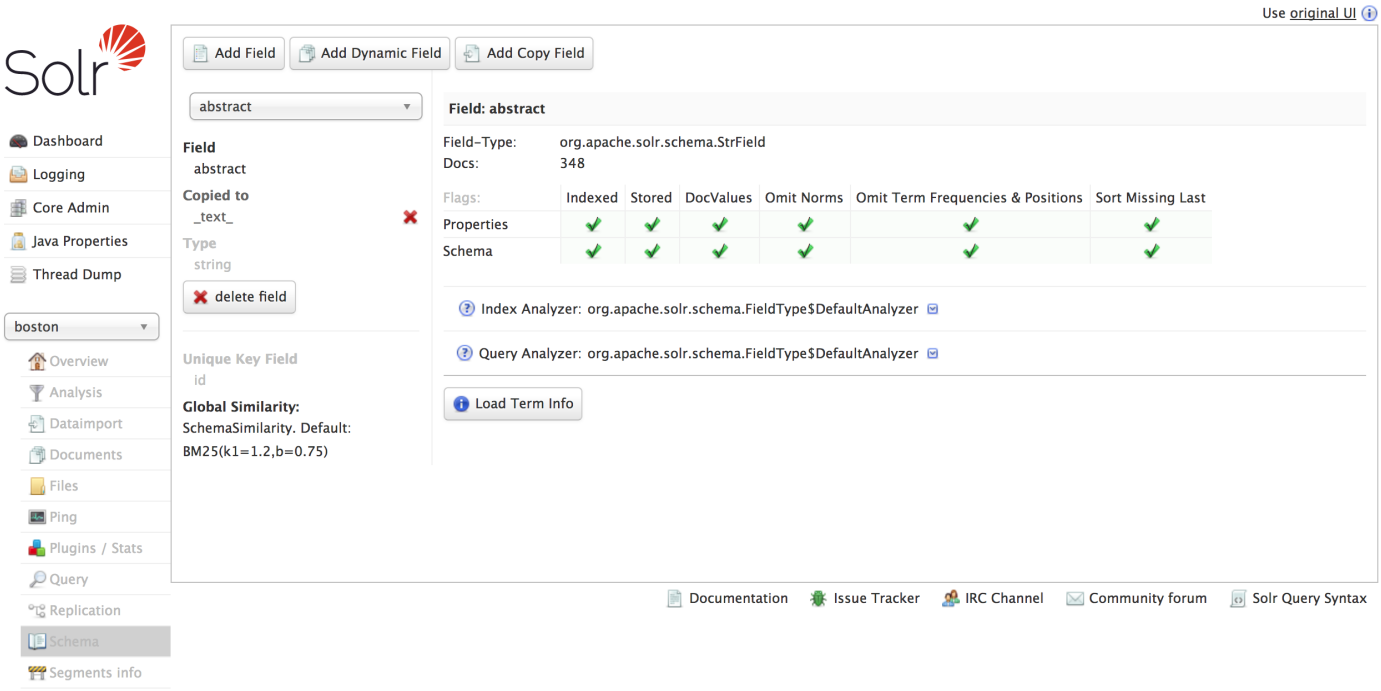
```
            "maxDistErr": "0.001",
            "distanceUnits": "degrees"
        }
    }
    requests.post(schema_url, json=location_rpt_quad_5m_payload)

    # create the DateRangeField type
    date_range_field_type_payload = {
        "add-field-type": {
            "name": "rdates",
            "class": "solr.DateRangeField",
            "multiValued": True
        }
    }
    requests.post(schema_url, json=date_range_field_type_payload)

    # now the other fields. The types are common and they are added by default to the
    fields = [
        {"name": "name", "type": "string"},
        {"name": "title", "type": "string"},
        {"name": "abstract", "type": "string"},
        {"name": "bbox", "type": "location_rpt_quad_5m"},
        {"name": "category", "type": "string"},
        {"name": "modified_date", "type": "rdates"},
        {"name": "username", "type": "string"},
        {"name": "keywords", "type": "string", "multiValued": True},
        {"name": "regions", "type": "string", "multiValued": True},
    ]

    headers = {
        "Content-type": "application/json"
    }

    for field in fields:
        data = {
            "add-field": field
        }
        requests.post(schema_url, json=data, headers=headers)

    print 'Schema generated.'
```

Now run the script using the Django shell. Before of that create a symbolic link to the foss4g_scripts directory:

```
$ cd /workshop/geonode/
ln -s ../foss4g_scripts .
$ python manage.py shell
```

```
>>> from foss4g_scripts import solr_schema
>>> solr_schema.create_schema()
```

Make sure the schema was generated by checking the schema endpoint at the Solr administrative site: http://localhost:8983/solr/#/boston/schema



Indexing data

The way you can pair Solr to GeoNode is by keeping in sync the GeoNode metadata for each layer, which are contained in a relational structure, to the Solr denormalized structure, based on documents. You will keep in sync the single metadata record for a given layer to the Solr document for that layer. In this tutorial you will keep in sync only some of the metadata, but once you master the concept the possibilities are limitless.

There is a number of way to load (index) data in Solr. Here you will use Python the Requests Python library to send POST JSON request to the Solr endpoint in order to create and update documents.

Create the Python script:

```
$ touch touch /workshop/foss4g_scripts/geonode2solr.py
```

Now add this Python code in geonode2solr.py:

```
import json
import requests
from geonode.layers.models import Layer


def layer2dict(layer):
    """
    Return a json representation for a GeoNode layer.
    """
    category = ''
    if layer.category:
        category = layer.category.gn_description
    wkt = "ENVELOPE({:f},{:f},{:f},{:f})".format(layer.bbox_x0, layer.bbox_x1, layer.
    layer_dict = {
                    'id': str(layer.uuid),
                    'name': layer.name,
                    'title': layer.title,
                    'abstract': layer.abstract,
                    'bbox': wkt,
                    'category': category,
                    'modified_date': layer.date.isoformat()[0:22] + 'Z',
                    'username': layer.owner.username,
                    'keywords': [kw.name for kw in layer.keywords.all()],
                    'regions': [region.name for region in layer.regions.all()],
                }
    print layer_dict
    return layer_dict


def layer_to_solr(layer):
    """
    Sync a layer in Solr.
    """

    layer_dict = layer2dict(layer)

    layer_json = json.dumps(layer_dict)

    url_solr_update = 'http://localhost:8983/solr/boston/update/json/docs'
    headers = {"content-type": "application/json"}
    params = {"commitWithin": 1500}
    res = requests.post(url_solr_update, data=layer_json, params=params,  headers=hea
    print res.json()


def sync():
    """
    Sync GeoNode layers with Solr.
    """
    for layer in Layer.objects.all():
```

```
        print 'Syncing layer %s to Solr' % layer.name
        layer_to_solr(layer)
```

Now run the script using the Django shell:

```
$ cd /workshop/geonode/
$ python manage.py shell
>>> from foss4g_scripts import geonode2solr
>>> geonode2solr.sync()
```

Check if data were added to Solr by checking this endpoint (more on this later):
http://localhost:8983/solr/boston/select?indent=on&q=:&wt=json

Indexing more data

A search engine is really useful when there are hundreds if not thousands of records to search. Your workshop GeoNode instance right now contains not even 10 layers (metadata records). Therefore for the aim of the tutorial you need a more extended instance with at least some hundreds of metadata.

For this purpose you will harvest metadata for a very large CSW catalogue using Python and OWSLib, and will index these metadata in Solr. The catalogue you are going to use for this purpose is Harvard Hypermap, which right now contains almost 200,000 layers metadata records. In case you are interested in Harvard Hypermap code base, you can check it out the CGA Harvard Hypermap github repository

You will harvest not all of the HHypermap layers, but just a part of them.

To make things more heterogeneous you will:

- add to each layer's metadata a random category (from the ones in GeoNode)
- use for each layer's metadata a random date (from 01-01-1970 to 12-01-2016)
- add to each layer's metadata up to 5 different regions (from the ones in GeoNode)
- add to each layer's metadata up to 5 different keywords (from an hard coded list)

Feel free to modify the script as you wish.

Create a Python script named *csw2solr.py*:

```
$ touch touch /workshop/foss4g_scripts/csw2solr.py
```

Add this Python code in the csw2solr script:

```
import json
import requests
import datetime
from random import randint
from owslib.csw import CatalogueServiceWeb
from geonode.base.models import TopicCategory, Region


def get_random_date():
    """ Get a random date between 01-01-1970 and 12-01-2016 """
    return datetime.date(randint(1970,2016), randint(1,12),randint(1,28))


def get_random_category():
    """ Get a random category from GeoNode """
    random_index = randint(0, TopicCategory.objects.all().count() - 1)
    tc = TopicCategory.objects.all()[random_index]
    return tc


def add_random_regions():
    """ Get up to 5 random regions from GeoNode """
    regions = []
    for i in range(0, randint(0, 5)):
        random_index = randint(0, Region.objects.all().count() - 1)
        region = Region.objects.all()[random_index]
        regions.append(region)
    return regions


def add_random_keywords():
    """ Get up to 5 random keywords """
    keywords_list = [
        "geology", "transportation", "utility", "agricolture",
        "idrology", "society", "biology", "industry", "USA",
        "environment", "pollution", "university", "research"
    ]
    keywords = []
    for i in range(0, randint(0, 5)):
        random_index = randint(0, len(keywords_list) - 1)
        keyword = keywords_list[random_index]
        keywords.append(keyword)
    return keywords

def layer2dict(layer):
    """
    Return a json representation for a csw layer.
    """
```

```
        wkt = "ENVELOPE({:f},{:f},{:f},{:f})".format(layer.bbox_x0, layer.bbox_x1, layer.
        layer_dict = {
                        'id': str(layer.uuid),
                        'name': layer.name,
                        'title': layer.title,
                        'abstract': layer.abstract,
                        'bbox': wkt,
                        'category': layer.category.gn_description,
                        'modified_date': layer.date.isoformat() + 'T00:00:00Z',
                        'keywords': [kw for kw in layer.keywords],
                        'regions': [region.name for region in layer.regions],
                    }
        print layer_dict
        return layer_dict


    def layer_to_solr(layer):
        """
        Sync a layer in Solr.
        """
        layer_dict = layer2dict(layer)
        layer_json = json.dumps(layer_dict)
        url_solr_update = 'http://localhost:8983/solr/boston/update/json/docs'
        headers = {"content-type": "application/json"}
        params = {"commitWithin": 1500}
        res = requests.post(url_solr_update, data=layer_json, params=params,  headers=hea
        print res.json()


    def sync():
        """
        Sync a bunch of csw layers with Solr.
        """

        class cswLayer(object):
            pass

        csw = CatalogueServiceWeb('http://hh.worldmap.harvard.edu/registry/hypermap/csw')
        count = 0
        for startposition in range(0, 2000, 10):
            csw.getrecords2(maxrecords=10, startposition=startposition)
            print csw.results
            for uuid in csw.records:
                record = csw.records[uuid]
                if record.bbox.crs.code == 4326:
                    layer = cswLayer()
                    layer.bbox_x0 = float(record.bbox.minx)
                    layer.bbox_y0 = float(record.bbox.miny)
                    layer.bbox_x1 = float(record.bbox.maxx)
                    layer.bbox_y1 = float(record.bbox.maxy)
                    if layer.bbox_x0 >= -180 and layer.bbox_x1 <= 180 and layer.bbox_y0 >
```

```
                    layer.uuid = uuid
                    layer.name = uuid
                    layer.title = record.title
                    layer.abstract = record.abstract
                    layer.url = record.source
                    layer.date = get_random_date()
                    layer.category = get_random_category()
                    layer.regions = add_random_regions()
                    layer.keywords = add_random_keywords()
                    layer_to_solr(layer)
                    print 'Imported record %s' % count
                    count += 1
    print 'Done.'
```

Now run the script using the Django shell:

```
$ cd /workshop/geonode/
$ python manage.py shell
>>> from foss4g_scripts import csw2solr
>>> csw2solr.sync()
```

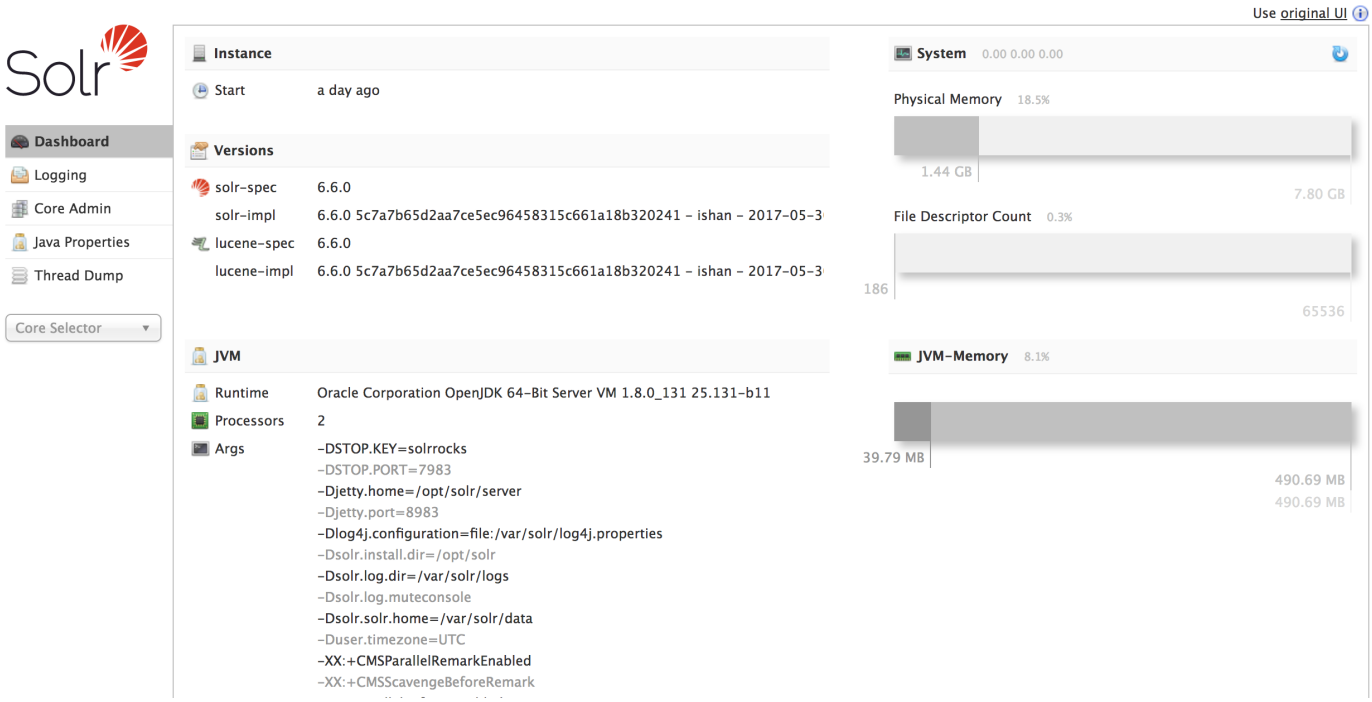
The number of documents in the *boston* Solr core now should be much larger. You can check it at: http://localhost:8983/solr/boston/select?indent=on&q=:&wt=json

Note: you may notice that even if index content more than once by running more time the Python scripts, it does not duplicate the results found. This is because the Solr schema for the *boston* core specifies a "uniqueKey" field called "id". Whenever you POST commands to Solr to add a document with the same value for the uniqueKey as an existing document, it automatically replaces it for you.

A quick tour of Solr administrative interface

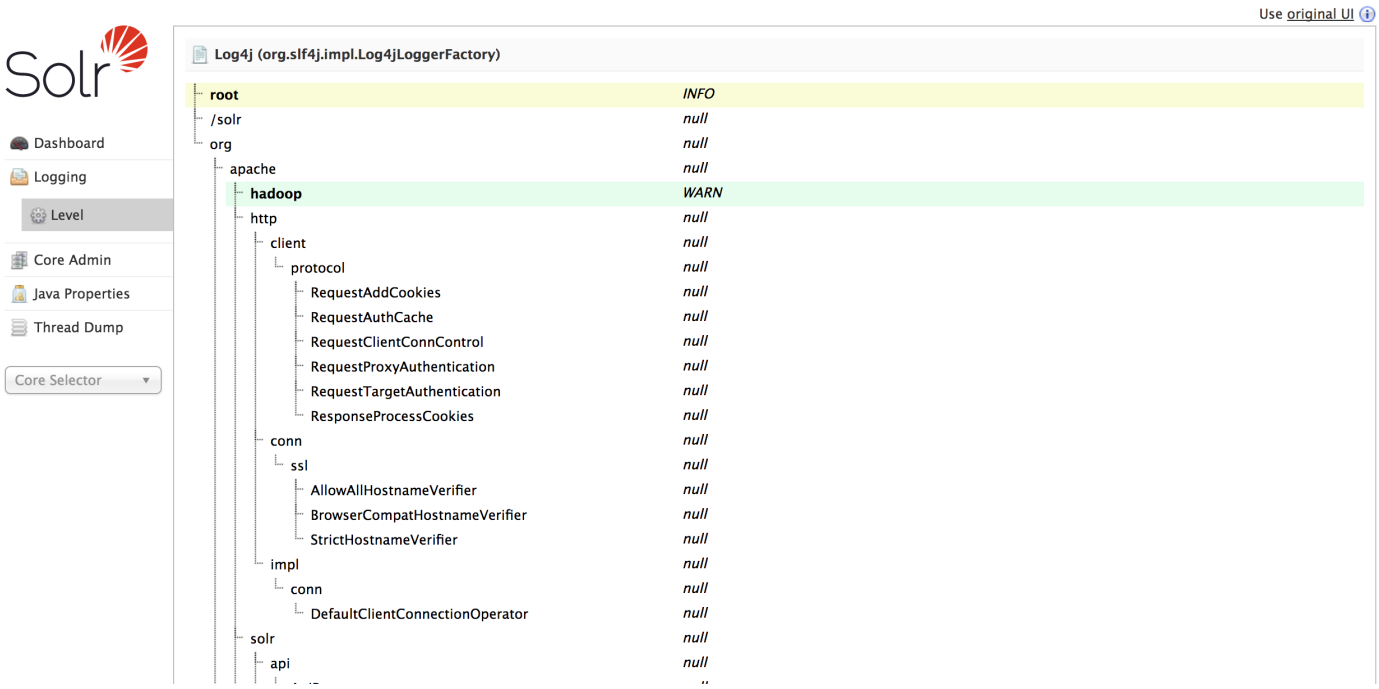
> Solr features a Web interface that makes it easy for Solr administrators and programmers to view Solr configuration details, run queries and analyze document fields in order to fine-tune a Solr configuration and access online documentation and other help.

Accessing the URL http://hostname:8983/solr/ will show the main **dashboard**. The dashboard displays information related to the Solr, Lucene and Java versions being used, some Java settings and information about the system (such as the physical and the JVM memory)

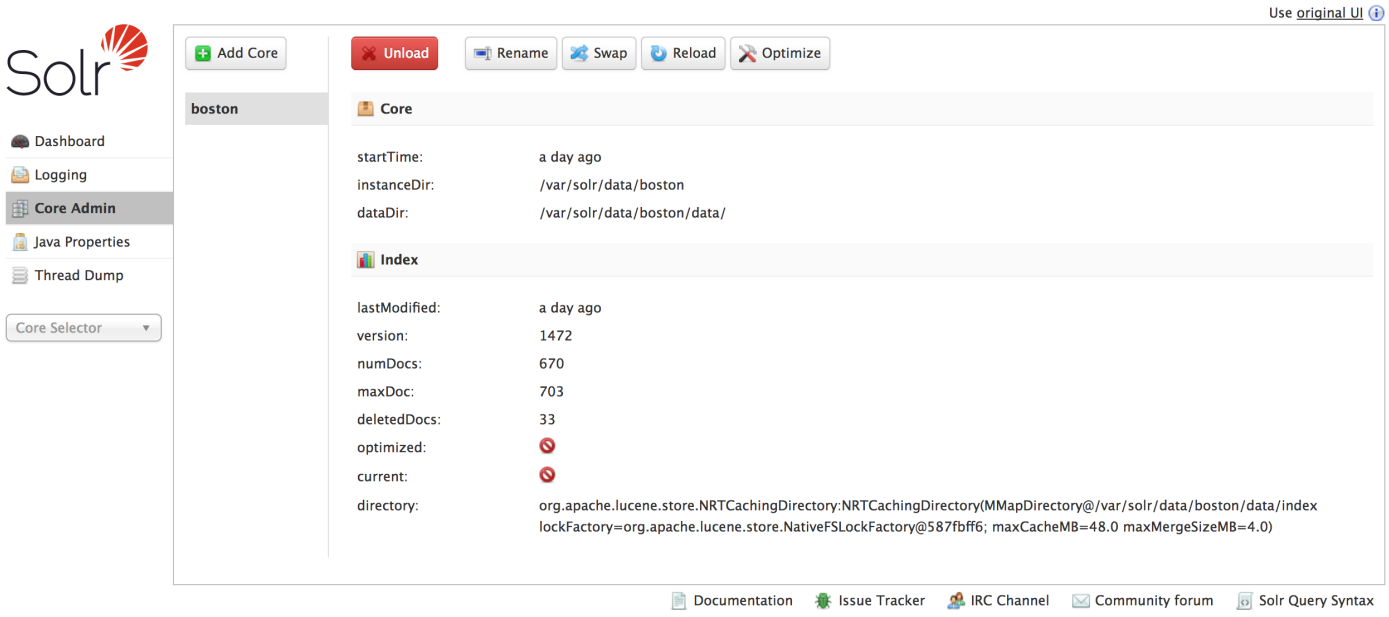


From the dashboard it is possible to navigate to other pages of the administrative interface. The **logging** page displays the latest log messages (which are accessible in more complete way using the tail or the less linux command in the shell)

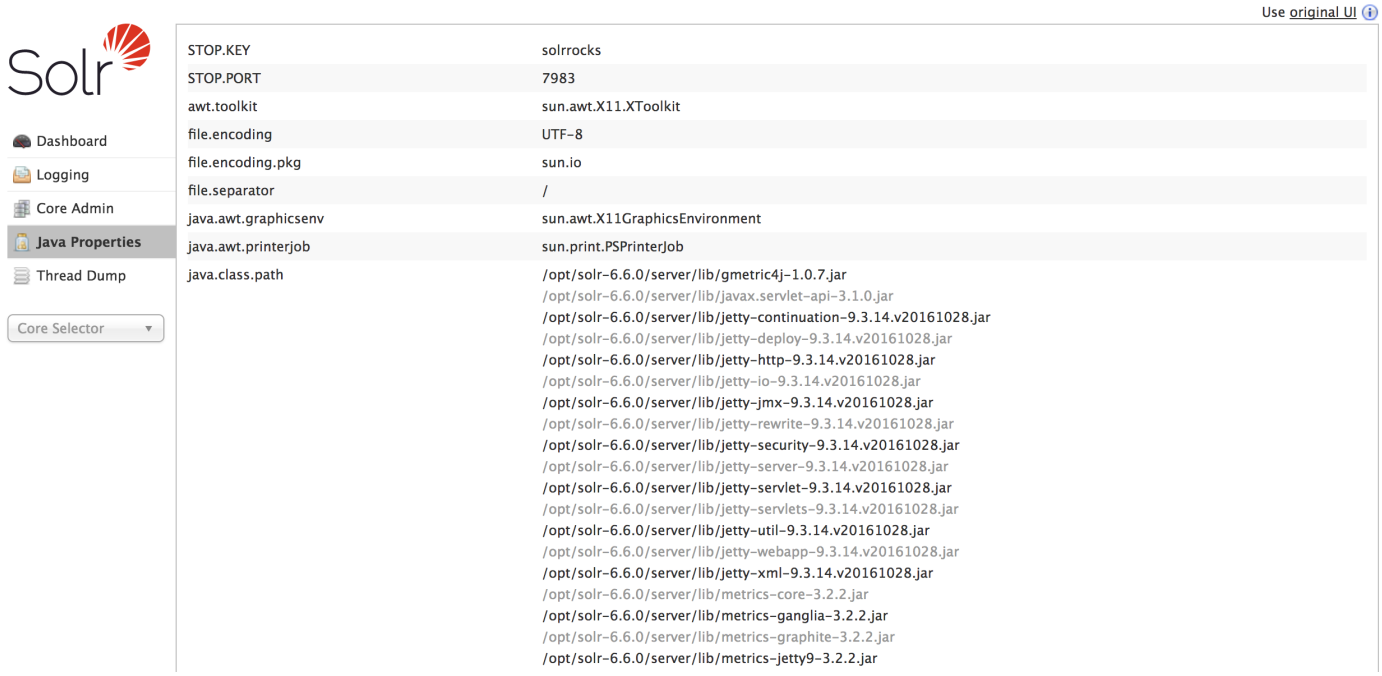
Clicking on the *level* menu it is possible to acces to the **Logging Level** page, from where it is possible to change the logging level



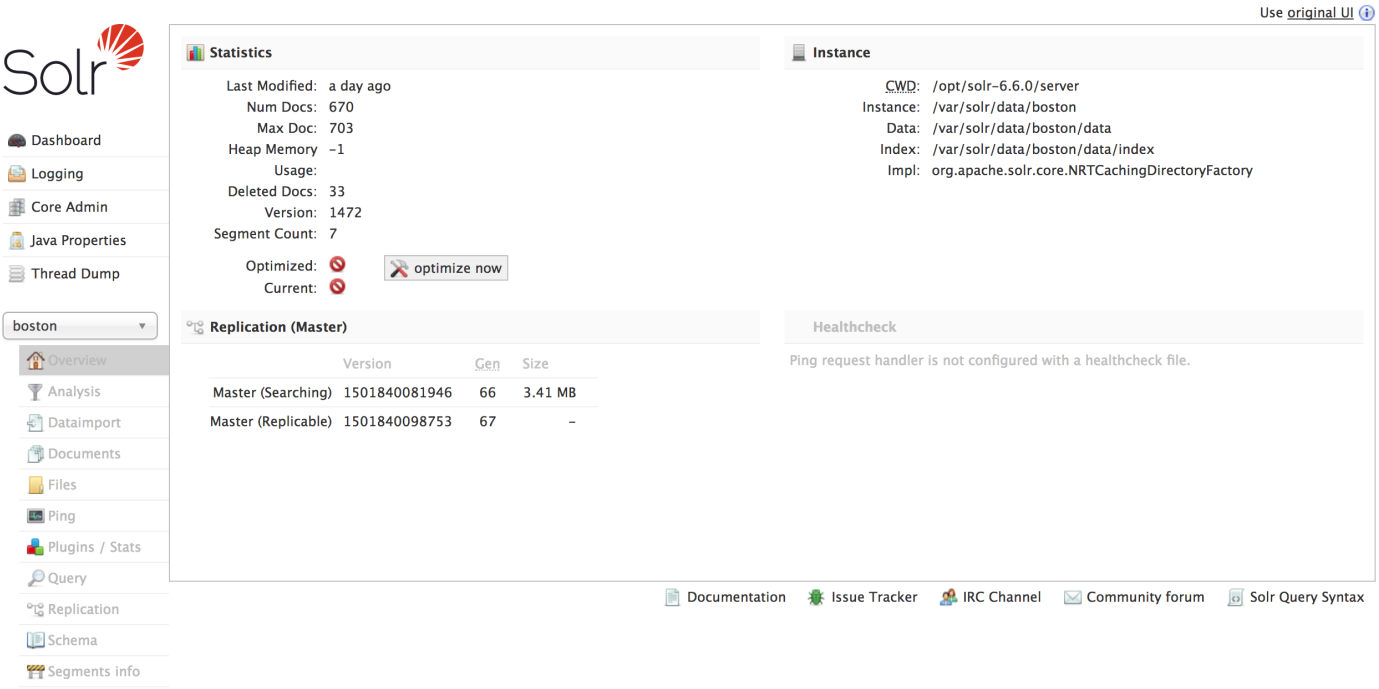
From the **Core Admin** pages it is possible to create new cores, or renaming and reloading existing ones. For each core it is possible to see the main related information, such as the data directory path, the number of indexed documents and the core start time



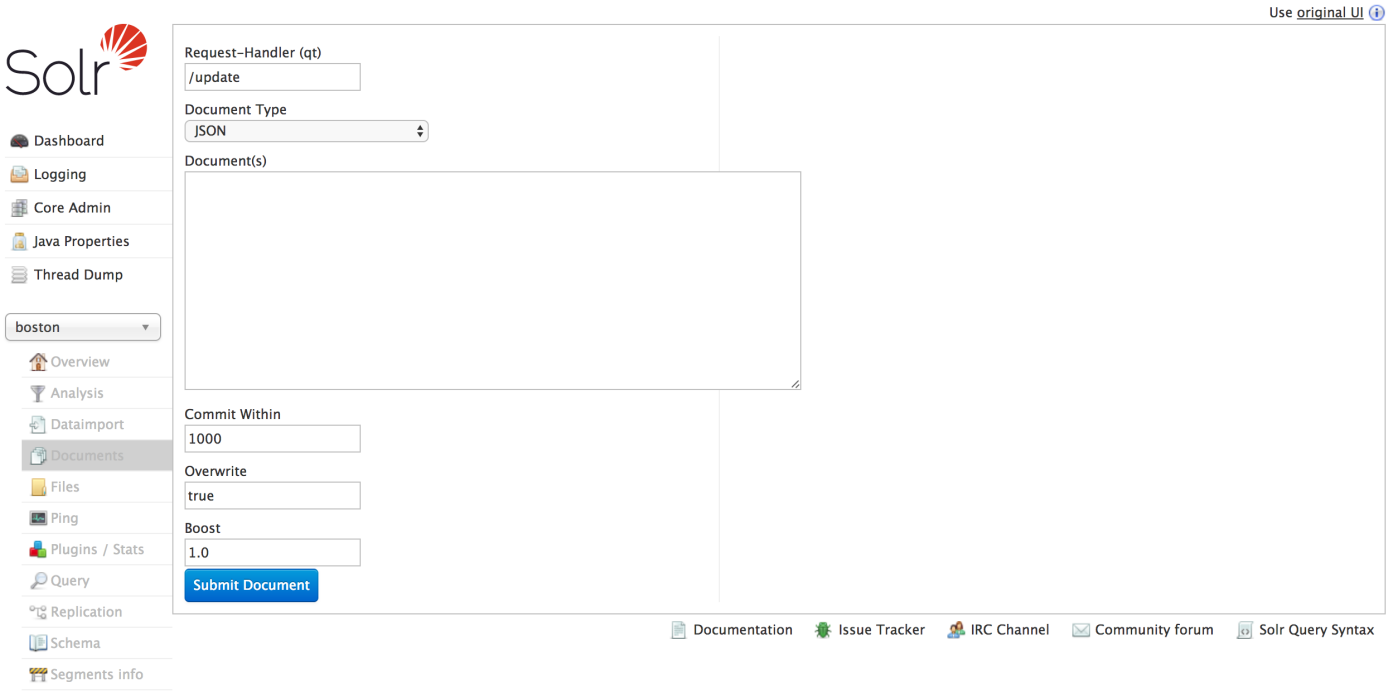
The **Java Properties** page displays the main information related to the Java environment: for example from here you can know which version of Java is being used, and which Java libraries are currently in the Java Path



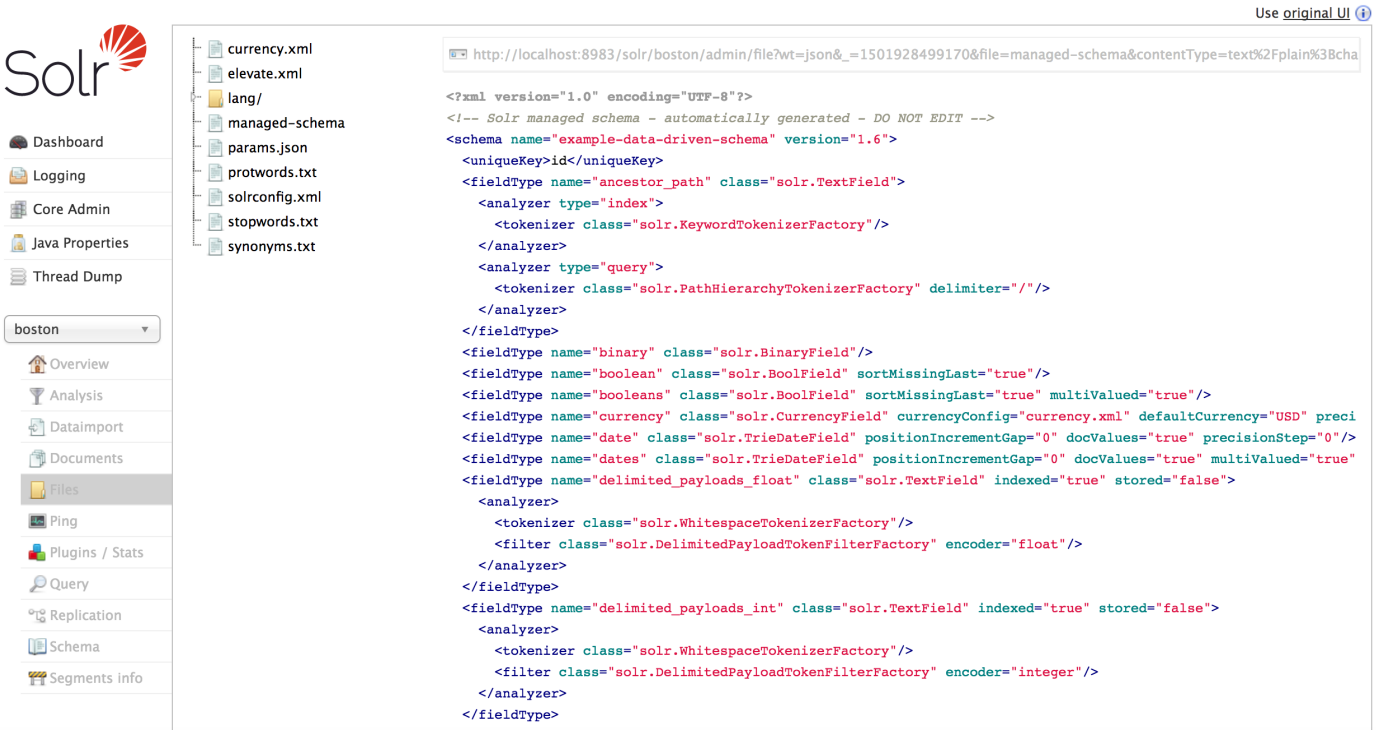
Using the **Core Overview** page, you can gain access to information related to a given core, for example the core data location and the number of indexed documents



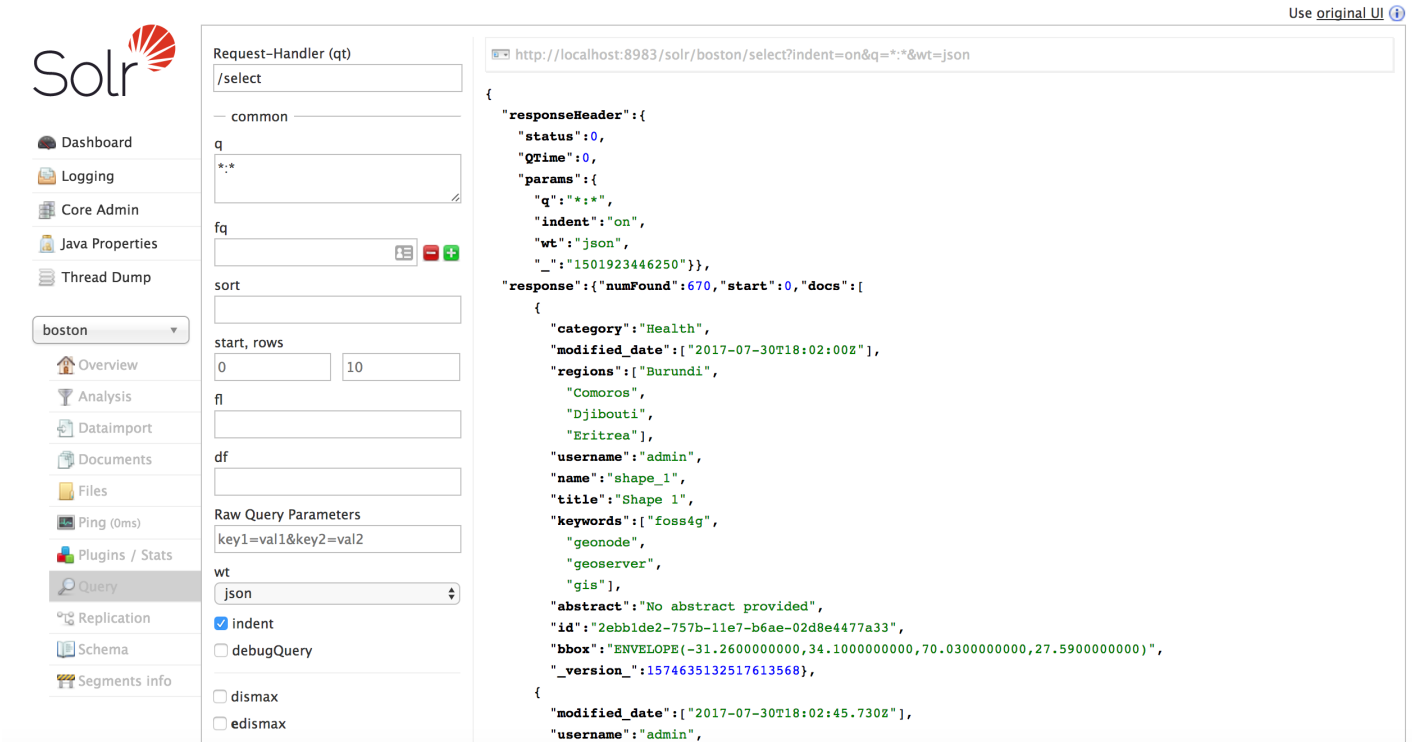
The **Core Documents** page provides a convenient form to index or update documents in Solr in different formats (JSON, XML, CSV...)



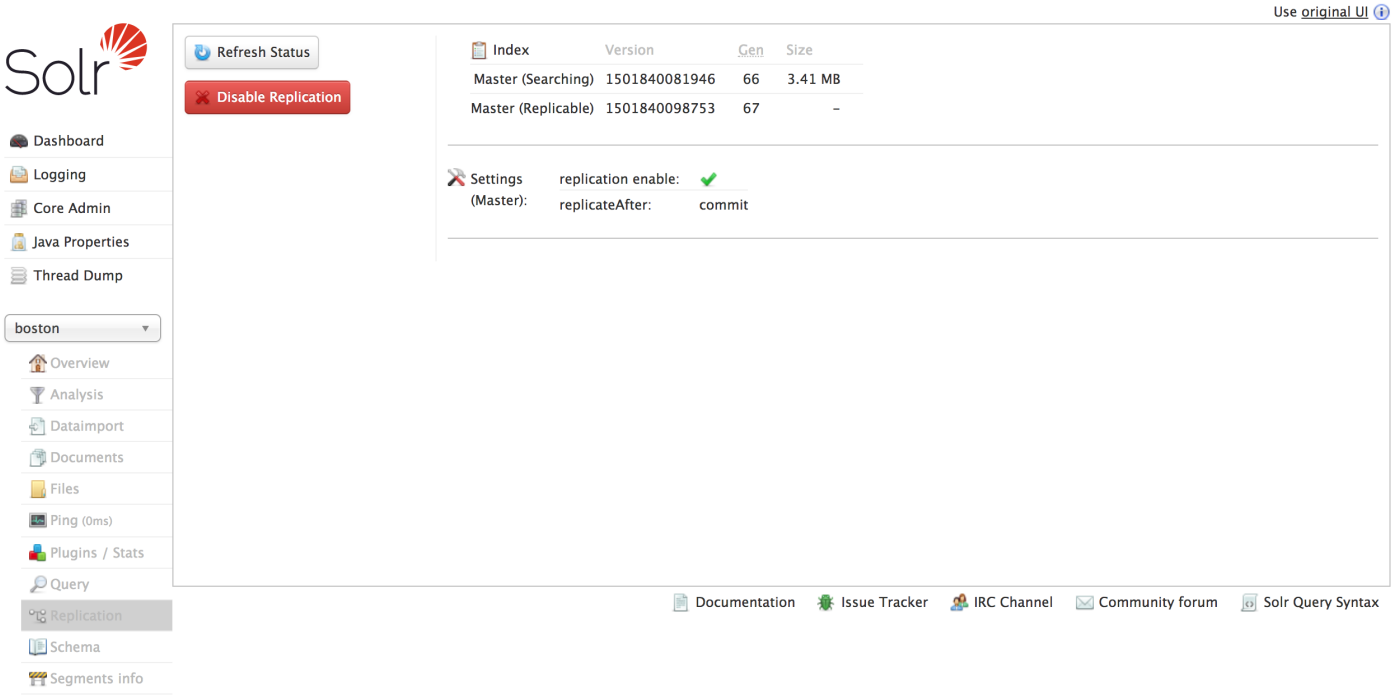
Core Files gives access to core configuration file (that can be anyway accessed from the shell in the server) such as the schema, the stopwords and the synonimus configuration files



Core Query page provides a very useful form that let you build query to the Solr core. From the form it is possible to insert query parameters such as the request handler ('/SELECT' by default), *q* (query string), *fq* (filer query), *sort* (sort field, which may be *ASC* or *DESC*), *start* and *rows* (needed for pagination), *fl* (fields list), *wt* (response format, default being XML), and many other parameters. By Clicking the *Execute Query* button the response will be displayed together with the convenient URL generated by the query builder



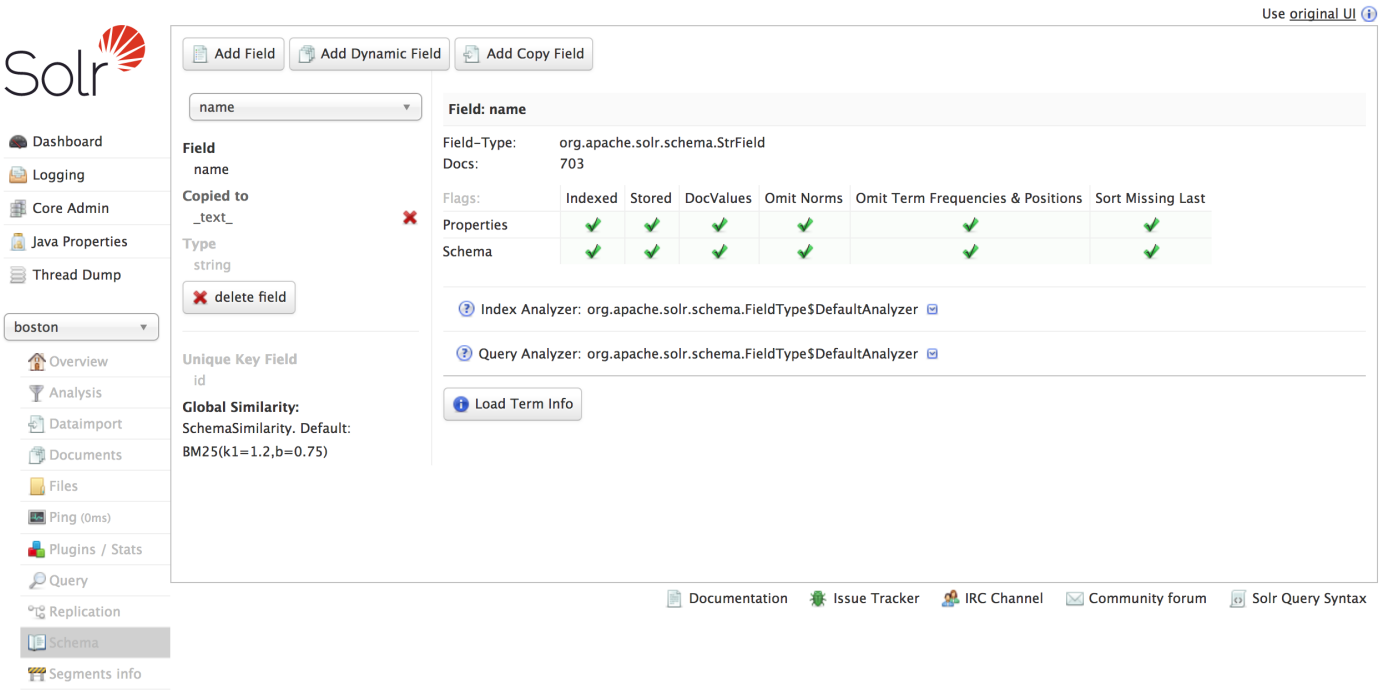
Core Replication page provides a page to configure Solr replication



Core Schema provides a form to add, modify and remove fields from the schema. When adding fields, it is possible to specify if the field is a regular, dynamic or copy field.

> A dynamic field is just like a regular field except it has a name with a wildcard in it. When you are indexing documents, a field that does not match any explicitly defined fields can be matched with a dynamic field.

> You might want to interpret some document fields in more than one way. Solr has a mechanism for making copies of fields so that you can apply several distinct field types to a single piece of incoming information. A common usage for this functionality is to create a single "search" field that will serve as the default query field when users or clients do not specify a field to query.



Querying Solr

Now that you uploaded a bunch of data to Solr, you can start sending queries and figure out which way it can be very powerful to pair a search engine to your portal.

You can test all of the following queries by directly clicking on the url, or by composing the queries using the Solr admin query builder: http://localhost:8983/solr/#/boston/query

Keyword matching

Get all of the field values of all of the records: q=*:*

http://localhost:8983/solr/boston/select?&q=*:*

This is what you will see:


```
{
  "responseHeader":{
    "status":0,
    "QTime":3,
```

```
    "params":{
      "q":"*:*",
      "indent":"on",
      "wt":"json",
      "_":"1501923446250"}},
  "response":{"numFound":670,"start":0,"docs":[
      {
        "category":"Health",
        "modified_date":["2017-07-30T18:02:00Z"],
        "regions":["Burundi",
          "Comoros",
          "Djibouti",
          "Eritrea"],
        "username":"admin",
        "name":"shape_1",
        "title":"Shape 1",
        "keywords":["foss4g",
          "geonode",
          "geoserver",
          "gis"],
        "abstract":"No abstract provided",
        "id":"2ebb1de2-757b-11e7-b6ae-02d8e4477a33",
        "bbox":"ENVELOPE(-31.2600000000,34.1000000000,70.0300000000,27.5900000000)",
        "_version_":1574635132517613568},
      {
        "modified_date":["2017-07-30T18:02:45.730Z"],
```


Have a look at this response: the response "responseHeader" section indicates the parameters used in the query and the response time used by Solr.

The "response" section indicates that there are 670 hits ("numFound":670), of which the first 10 were returned, since by default Solr uses start=0 and rows=10 as the pagination parameters. You can specify these params to page through results, where start is the (zero-based) position of the first result to return, and rows is the page size.

For example you can paginate records results starting from the 50th till the 70th using: rows=20&start=50 (rows is 10 by default)

http://localhost:8983/solr/boston/select?indent=on&q=:&rows=20&start=50

You can use the *wt* parameter to get all of the records in a given format, for example json (wt by default is xml):

http://localhost:8983/solr/boston/select?indent=on&q=:&wt=json

This is how you can query a field with an exact text string: q=title:"Landscape condition"

http://localhost:8983/solr/boston/select?indent=on&q=title:Landscape%20condition&wt=json

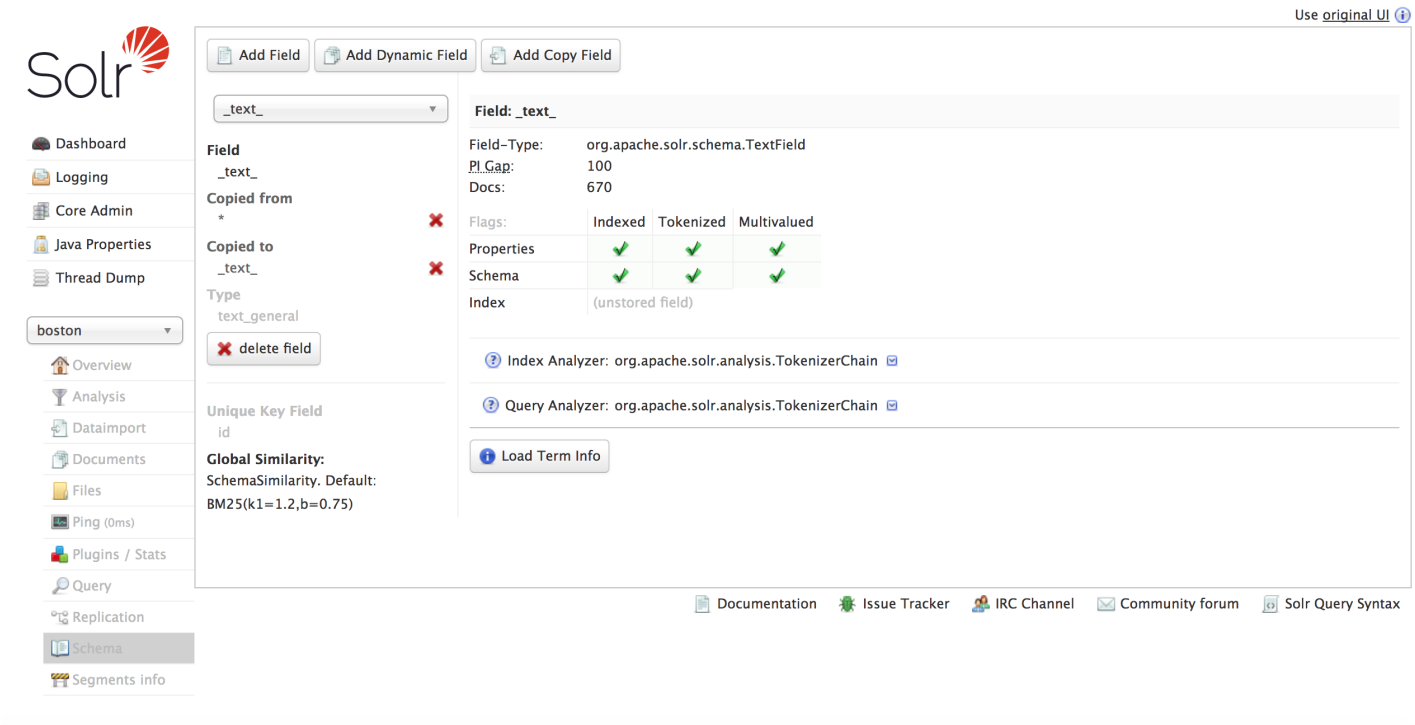
To query a field with a wildcard: q=abstract:Natura*

http://localhost:8983/solr/boston/select?indent=on&q=abstract:Natura*&wt=json

Full text query with more parameters: q=*text*:soil% AND *text*:database

http://localhost:8983/solr/boston/select?indent=on&q=*text*:soil%20and%20*text*:database&rows=10&start=0&wt=json

Note: by default Solr creates a *_text_* copy field where a copy of all of the other fields is appended. This makes it as a convenient field to use for string searching



You can use the *fl* parameter (fields list) to return only some of the fields, for example: fl=title, abstract

http://localhost:8983/solr/boston/select?fl=title,%20abstract&indent=on&q=:&rows=10&start=0&wt=json

Query a multiValued field (fields which can store arrays of values): regions:Antarctica OR regions:Asia

http://localhost:8983/solr/boston/select?indent=on&q=regions:Antarctica%20or%20regions:Asia&rows=10&start=0&wt=json

Use the *sort* parameter to sort by a field (you need to specify *asc* or *desc* to specify sort order): q=:&sort=title asc

http://localhost:8983/solr/boston/select?indent=on&q=:&sort=title%20asc&wt=json

Scored results

Solr can return results with a score, calculated by Lucene. Score indicates how much a given document is relevant for a user. For getting this you need to specify *score* in the *fl* parameter:

http://localhost:8983/solr/boston/select?fl=*,score&indent=on&q=*text*:geographic&rows=10&start=0&wt=json

Proximity matching

Lucene supports finding words that are within a specific distance away. For exmaple, to search for “service information” within 4 words from each other you can do like this: *text*:“service,information”~4

http://localhost:8983/solr/boston/select?indent=on&q=*text*:%22service%20information%22~4&rows=10&start=0&wt=json

Boosts

Query-time boosts allow one to specify which terms/clauses are “more important”. The higher the boost factor, the more relevant the term will be, and therefore the higher the corresponding document scores.

A typical boosting technique is assigning higher boosts to title matches than to body content matches:

(title:service OR title:information)^1.5 (abstract:service OR abstract:information)

http://localhost:8983/solr/boston/select?indent=on&q=(title:*service*%20OR%20title:*information*)^1.5%20(abstract:*service*%20OR%20abstract:*information*)&rows=10&start=0&wt=json

Date ranges

Solr provides a very powerful and convenient syntax to query dates and date ranges. If you want to get documents having a date field in a given date range here is how you can do it: q=modified_date:[1970-01 TO 1990-12]

http://localhost:8983/solr/boston/select?indent=on&q=modified_date:[2016-11%20TO%202016-12]&wt=json

Date range syntax can be more complex:

http://localhost:8983/solr/boston/select?indent=on&q=modified_date:[*%20TO%202016-11]&wt=json

http://localhost:8983/solr/boston/select?indent=on&q=modified_date:[2016-11%20TO%20NOW]&wt=json

http://localhost:8983/solr/boston/select?indent=on&q=modified_date:[2016-11%20TO%20NOW-10DAY]&wt=json

Spatial Search

Solr provides native spatial support and query syntax, thanks to its underlying engine based on Spatial4j and JTS Topology Suite.

There are different spatial field types in Solr, and all of them let to search documents given a spatial extent. Some of the spatial fields can even be used to provide a spatial representation: for example GeoServer with the Solr plugin makes possible to publish documents directly from Solr.

There are two different filters to search documents by extent. You will use both of them to search the *boston* Solr core using the *bbox* spatial field.

geofilt spatial filter

The *geofilt* filter allows you to retrieve results based on the geospatial distance from a given point (a buffer around a point). For example, to find all documents within five kilometers of a given lat/lon point, you could enter &q=:&fq={!geofilt sfield=bbox}&pt=45.15,-93.85&d=5

http://localhost:8983/solr/boston/select?d=1&indent=on&wt=json&q=:&fq={!geofilt%20sfield=bbox}&pt=-71.10,42.37&d=5

bbox spatial filter

The *bbox* filter is very similar to *geofilt* except it uses the bounding box of the calculated circle

http://localhost:8983/solr/boston/select?d=1&indent=on&wt=json&q=:&fq={!bbox%20sfield=bbox}&pt=-71.10,42.37&d=5

filtering by arbitrary extent

It is easy to filter Solr documents by an arbitrary extent: &q=:&fq=store:[45,-94%20TO%2046,-93]

http://localhost:8983/solr/boston/select?d=1&indent=on&wt=json&q=:&fq=bbox:[-71,42%20TO%20-70,43]

Faceting

> *Faceting* is the arrangement of search results into categories based on indexed terms. Searchers are presented with the indexed terms, along with numerical counts of how many matching documents were found were each term. Faceting makes it easy for users to explore search results, narrowing in on exactly the results they are looking for.

Faceting can be based on keywords, but also on dates and spatial extents.

Keyword facets

In the context of an SDI portal such as with GeoNode, *keyword facets* can be useful to provide an immediate indication of the number of matching layers and maps for each given tag, category or region.

Number of layers for each given keyword:

http://localhost:8983/solr/boston/select?facet.field=keywords&facet=on&indent=on&q=:&rows=10&wt=json


```
  },
- facet_counts: {
    facet_queries: { },
  - facet_fields: {
    - keywords: [
          "biology",
          130,
          "geology",
          124,
          "idrology",
          121,
          "industry",
          121,
          "transportation",
          119,
          "USA",
          116,
          "pollution",
          115,
          "agricolture",
          104,
          "society",
          100,
          "research",
          99,
          "university",
          99,
          "utility",
```


Number of layers for each given region:

http://localhost:8983/solr/boston/select?facet.field=regions&facet=on&indent=on&q=:&rows=10&wt=json

```
,,
- facet_counts: {
    facet_queries: { },
  - facet_fields: {
      - regions: [
            "Turks and Caicos Islands",
            16,
            "Iran",
            13,
            "Madagascar",
            13,
            "Chile",
            12,
            "Finland",
            12,
            "Niger",
            12,
            "Poland",
            12,
            "Zimbabwe",
            12,
            "Asia",
            11,
            "Central Africa",
            11,
            "French Guiana",
            11,
            "Pacific",
            11,
```


Number of layers for each given category:

http://localhost:8983/solr/boston/select?facet.field=category&facet=on&indent=on&q=:&rows=10&wt=json


```
,,
- facet_counts: {
    facet_queries: { },
  - facet_fields: {
      - category: [
            "Elevation",
            46,
            "Inland Waters",
            46,
            "Intelligence Military",
            45,
            "Biota",
            42,
            "Transportation",
            42,
            "Farming",
            39,
            "Health",
            37,
            "Society",
            36,
            "Utilities Communication",
            36,
            "Environment",
            34,
            "Boundaries",
            33,
            "Oceans",
            33,
            "Planning Cadastre",
            31,
```


Temporal facets

temporal facets can be useful to provide users an indication of the number of layers and maps with metadata within given temporal ranges. For example to provide a geo portal with an histogram which tracks how many layers are available in the portal with a representational date falling within each month, year or century.

Number of layers per year in the last 50 years:

http://localhost:8983/solr/boston/select?q=:&facet.range=modified_date&facet=true&facet.range.start=NOW-

50YEAR&facet.range.end=NOW&facet.range.gap=%2B1YEAR&wt=json


```
- facet_counts: {
    facet_queries: { },
    facet_fields: { },
  - facet_ranges: {
      - modified_date: {
          - counts: [
                "1967-08-05T12:53:07.068Z",
                0,
                "1968-08-05T12:53:07.068Z",
                0,
                "1969-08-05T12:53:07.068Z",
                6,
                "1970-08-05T12:53:07.068Z",
                12,
                "1971-08-05T12:53:07.068Z",
                5,
                "1972-08-05T12:53:07.068Z",
                6,
                "1973-08-05T12:53:07.068Z",
                15,
                "1974-08-05T12:53:07.068Z",
                6,
                "1975-08-05T12:53:07.068Z",
                4,
                "1976-08-05T12:53:07.068Z",
                11,
                "1977-08-05T12:53:07.068Z",
```


Number of layers per month in the last 50 years:

http://localhost:8983/solr/boston/select?q=:&facet.range=modified_date&facet=true&facet.range.start=NOW-50YEAR&facet.range.end=NOW&facet.range.gap=%2B1MONTH&wt=json

Spatial facets

Solr spatial field supports generating a 2D grid of facet counts for documents having spatial data in each grid cell. For high-detail grids, this can be used to plot points, and for lesser detail it can be used for heatmap generation.

http://localhost:8983/solr/boston/select?q=:&facet=true&facet.heatmap=bbox&facet.heatmap.geom[“-180%20-90”%20TO%20”180%2090”]&wt=json

```
- facet_counts: {
    facet_queries: { },
    facet_fields: { },
    facet_ranges: { },
    facet_intervals: { },
  - facet_heatmaps: {
     - bbox: [
          "gridLevel",
          4,
          "columns",
          16,
          "rows",
          16,
          "minX",
          -180,
          "maxX",
          180,
          "minY",
          -180,
          "maxY",
          180,
          "counts_ints2D",
        - [
            null,
            null,
            null,
          + […],
          - [
              126,
              130,
              128,
              127,
              126,
              129,
              126,
              128,
              126,
              125,
```


> The minX, maxX, minY, maxY reports the region where the counts are. This is the minimally enclosing bounding rectangle of the input geom at the target grid level. The columns and rows values are how many columns and rows that the output rectangle is to be divided by evenly. The counts_ints2D key has a 2D array of integers. The initial outer level is in row order (top-down), then the inner arrays are the columns (left-right). If any array would be all zeros, a null is returned instead for efficiency reasons. The entire value is null if there is no matching spatial data.

Heatmap is one way to facet spatial data. The other way is concentric circles of distance, which can be obtained using multiple facet.query params.

foss4g_2017_geonode_solr

Developing a custom application for GeoNode

In this tutorial you will learn how to create a custom Django application and use it in GeoNode.

The application you will create is intended to query Solr to provide the end users with a metadata full text search tool. The application will have a web page with a form, from where in a text box the user will be able to enter a search string. When submitting the form a query will be sent to Solr and the first 10 results will be displayed in the page.

Each result will contain the main metadata record information (title, abstract, category...) and its score.

Create the Django application

As a first thing you need to create the Django application, which you will name *solr*. For this purpose open a shell window, activate the virtual environment and create the application using the *startapp* option of the Django *manage.py* command:

```
$ . /workshop/env/bin/activate
$ cd /workshop/geonode
$ mkdir geonode/solr
$ python manage.py startapp solr geonode/solr
```

Create a template

You will create a Django template containing the user interface of the application. The user interface will have a form to enter the search string and a list of results.

> Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

> A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships built-in backends for its own template system,

> creatively called the Django template language (DTL), and for the popular alternative Jinja2. Backends for other template languages may be available from third-parties.

Create the template:

```
$ mkdir -p geonode/sol/templates/solr
$ touch geonode/solr/templates/solr/index.html
```

Add this html code in the *geonode/solr/index.html* template.

Create the url dispatcher

To design URLs for an application in Django, you create a Python module informally called a URL configuration, which is named *urls.py*

GeoNode has a main *urls.py* URL configuration at geonode/urls.py The recommended approach for a Django application is to have its own URL configuration, and import it from the main URL configuration.

Therefore you are going to import the Solr application URL configuration (which will be in a file *geonode/solr/urls.py*) from the GeoNode main URL configuration.

For this purpose, add this code at the end of *geonode/urls.py*

```
# solr application urls
urlpatterns += patterns('',
    url(r'^solr/', include('geonode.solr.urls')),
)
```

Now create the Solr application URL configuration file:

```
$ touch geonode/solr/urls.py
```

Add this code in *geonode/solr/urls.py*:

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

urlpatterns = patterns('geonode.solr.views',
                       url(r'^$', 'index', name='index'),
                       )
```

There is only one URL that the user can use in the solr application: its path will be http://localhost:8000/solr/

Add the application in INSTALLED_APPS

Each Django application that is intended to use in a project must be added in the INSTALLED_APPS Django setting.

The main GeoNode *settings.py* file is at *geonode/settings.py* and contains the following applications in the INSTALLED_APPS setting:

```
GEONODE_APPS = (
    # GeoNode internal apps
    'geonode.people',
    'geonode.base',
    'geonode.layers',
    'geonode.maps',
    'geonode.proxy',
    'geonode.security',
    'geonode.social',
    'geonode.catalogue',
    'geonode.documents',
    'geonode.api',
    'geonode.groups',
    'geonode.services',

    # QGIS Server Apps
    # 'geonode_qgis_server',

    # GeoServer Apps
    # Geoserver needs to come last because
    # it's signals may rely on other apps' signals.
    'geonode.geoserver',
    'geonode.upload',
    'geonode.tasks',

)

GEONODE_CONTRIB_APPS = (
    # GeoNode Contrib Apps
    'geonode.contrib.dynamic',
    'geonode.contrib.exif',
    'geonode.contrib.favorite',
    'geonode.contrib.geogig',
    'geonode.contrib.geosites',
    'geonode.contrib.nlp',
    'geonode.contrib.slack',
    'geonode.contrib.metadataxsl'
```

```
)

# Uncomment the following line to enable contrib apps
# GEONODE_APPS = GEONODE_APPS + GEONODE_CONTRIB_APPS

INSTALLED_APPS = (

    'modeltranslation',

    # Boostrap admin theme
    # 'django_admin_bootstrapped.bootstrap3',
    # 'django_admin_bootstrapped',

    # Apps bundled with Django
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.sitemaps',
    'django.contrib.staticfiles',
    'django.contrib.messages',
    'django.contrib.humanize',
    'django.contrib.gis',

    # Third party apps

    # Utility
    'pagination',
    'taggit',
    'treebeard',
    'friendlytagloader',
    'geoexplorer',
    'leaflet',
    'django_extensions',
    #'geonode-client',
    # 'haystack',
    'autocomplete_light',
    'mptt',
    # 'modeltranslation',
    # 'djkombu',
    'djcelery',
    # 'kombu.transport.django',
    'storages',

    # Theme
    "pinax_theme_bootstrap_account",
    "pinax_theme_bootstrap",
    'django_forms_bootstrap',

    # Social
```

```
        'account',
        'avatar',
        'dialogos',
        'agon_ratings',
        # 'notification',
        'announcements',
        'actstream',
        'user_messages',
        'tastypie',
        'polymorphic',
        'guardian',
        'oauth2_provider',

    ) + GEONODE_APPS
```

You will customize the INSTALLED_APPS setting by changing its value in *geonode/local_settings.py*. At the end of that file add the following lines:

```
from django.conf import settings
settings.INSTALLED_APPS = settings.INSTALLED_APPS + ('geonode.solr',)
LOCAL_ROOT = os.path.abspath(os.path.dirname(__file__))
settings.TEMPLATES[0]['DIRS'].insert(0, os.path.join(LOCAL_ROOT, "solr/templates"))
```

Create the view

> A Django view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path.

Here you will create the view which will process the user request. The view will get the query string from the web request, and then will send a query to Solr using the Requests Python library.

Then the view will process the JSON response from Solr, and send the results (docs) dictionary and the count of the results to the template.

Add this code to create the view in *geonode/solr/views.py*:

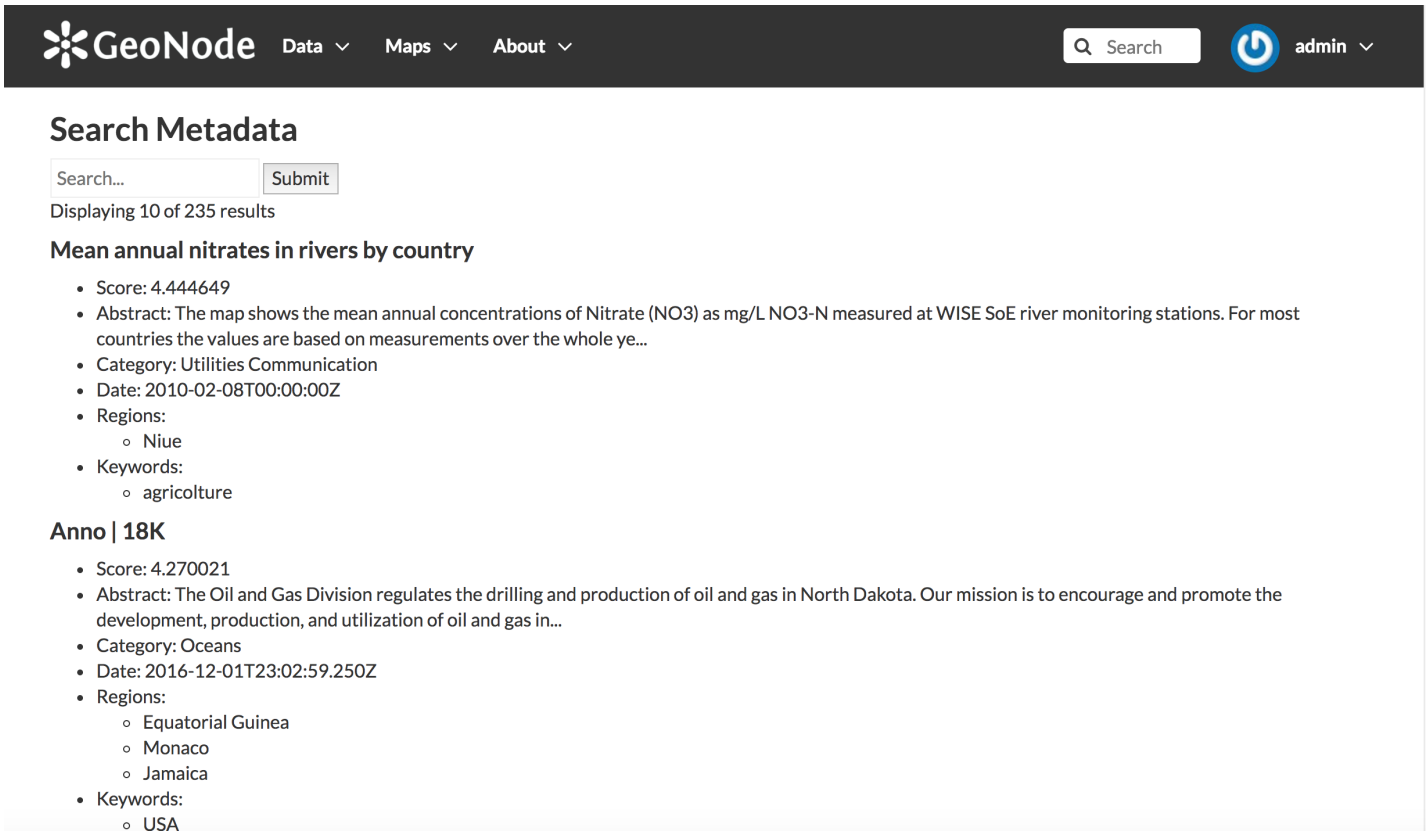
```
import json, requests
from django.shortcuts import render_to_response
```

```
from django.template import RequestContext


def index(request):
    '''
    A view to search metadata in Solr.
    '''

    search_text = '*'
    if request.method == 'GET':
            if 'search_text' in request.GET:
                search_text = request.GET.get('search_text', None)
    if not search_text:
        search_text = '*'

    url = 'http://localhost:8983/solr/boston/select'
    params = dict(
        q='_text_:%s' % search_text,
        fl='*,score',
        wt='json'
    )
    resp = requests.get(url=url, params=params)
    data = json.loads(resp.text)

    count = data['response']['numFound']
    docs = data['response']['docs']
    return render_to_response('solr/index.html',
                              RequestContext(request,
                                  {
                                      'docs': docs,
                                      'count': count,
                                  },
                              )
                          )
```

Run the application

Now you should be ready to run the application. Make sure that GeoNode is correctly running, and then check if the application endpoint is operational: http://localhost:8000/solr/

If everything works, you should see the Solr's Django application rendered. Try to search metadata with any text string you like



Bonus steps

Solr pagination

As a bonus step, if you are feeling adventurous, try to implement Solr pagination.

Hint: you will need to read the *start* value in *views.py* from *data['response']* and pass it in the *RequestContext* dictionary, together with *docs* and *count*.

GeoNode Project

In this workshop you are customizing GeoNode by adding code files directly in the GeoNode directory cloned from GitHub. While this can be the fastest approach, it has a big limitation: at some point you will edit files from the GeoNode code base (for example you have edited *geonode/ulrs.py*) which will introduce a lot of problems when updating to a new version of GeoNode.

To avoid this problems, the recommended approach is to use a geonode-project: https://github.com/GeoNode/geonode-project

After forking your geonode-project, you will be able to customize GeoNode without touching any of the GeoNode core files. Therefore the GeoNode update procedure will be much easier.

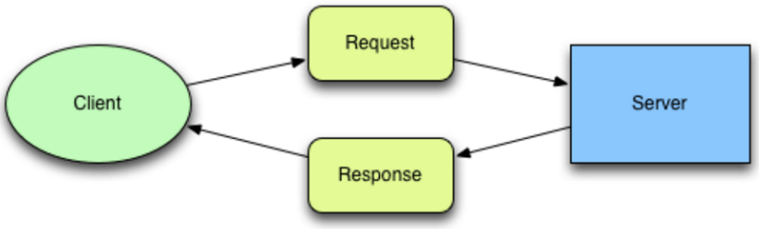
foss4g_2017_geonode_solr

Running asynchronous tasks using a task queue (Celery/RabbitMQ)

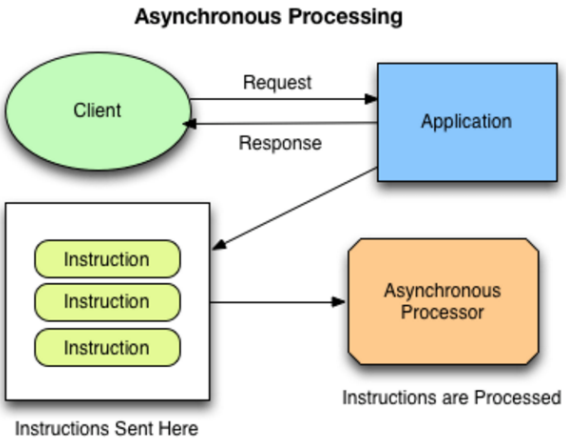
The HTTP request/response cycle can be kept synchronous as far as there are very quick interactions (few milliseconds) between the client and the servers.

Unfortunately there are cases when the cycle become slower and slower because of some time consuming tasks (1, 2 seconds or even more...): in these situations the best practice for a web application is to process asynchronously these tasks using a task queue.

Furthermore some task must be scheduled, or need to interact with external services, which can take time. In those cases we run these longer tasks separately, in different processes.



HTTP Request Response Cycle



Images from http://blog.codepath.com/

Task queues

Typical uses cases in a web application where a task queue is the way to go:

- Asynchronous tasks such as:
  - Thumbnails generation
  - Notifications

    - Data denormalization
    - Search index synchronization
- Replacing cron jobs
    - Backups
    - Maintenance jobs
    - pdf reports

In the case of a geoportal platform as GeoNode the use cases can be extended to:

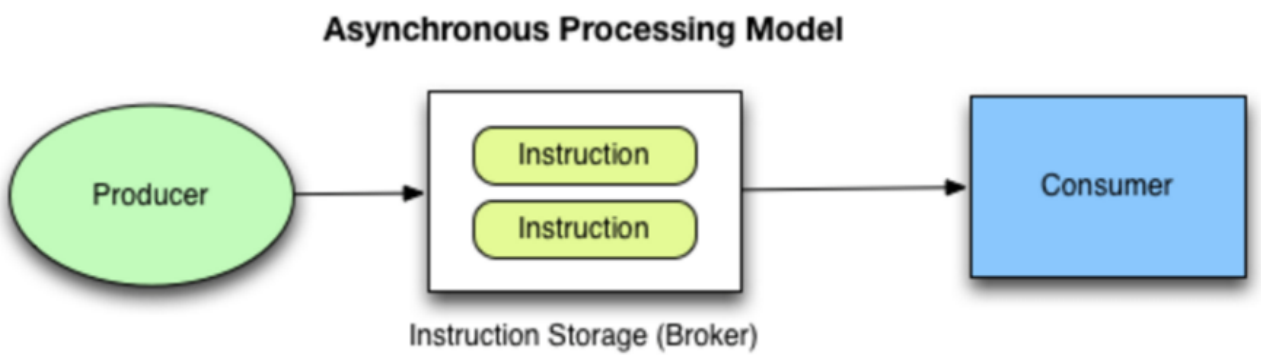
- Upload a shapefile to the server (next generation GeoNode)
- OGC services harvesting (Harvard Hypermap)
- GeoProcessing operations
- Geodata maintenance

Asynchronous processing model

The asynchronous processing model is composed by services that create processing tasks and by services which consume and process these tasks accordingly.

A **message queue** is a **broker** which facilitates message passing by providing a protocol or interface which other services can access. In the context of a web application as GeoNode the **producer** is the client application that creates messages based on the user interaction (for example a user that saves metadata, and sends a synchronization process to the search engine). The **consumer** is a daemon process (**Celery** in the case of GeoNode) that can consume the messages and run the needed process.

A more complex use case is when there are two or more applications which are the producer and consumer of the messages. For example the user of the web application, written in Django, could produce a message which is consumed by a daemon of an another program.



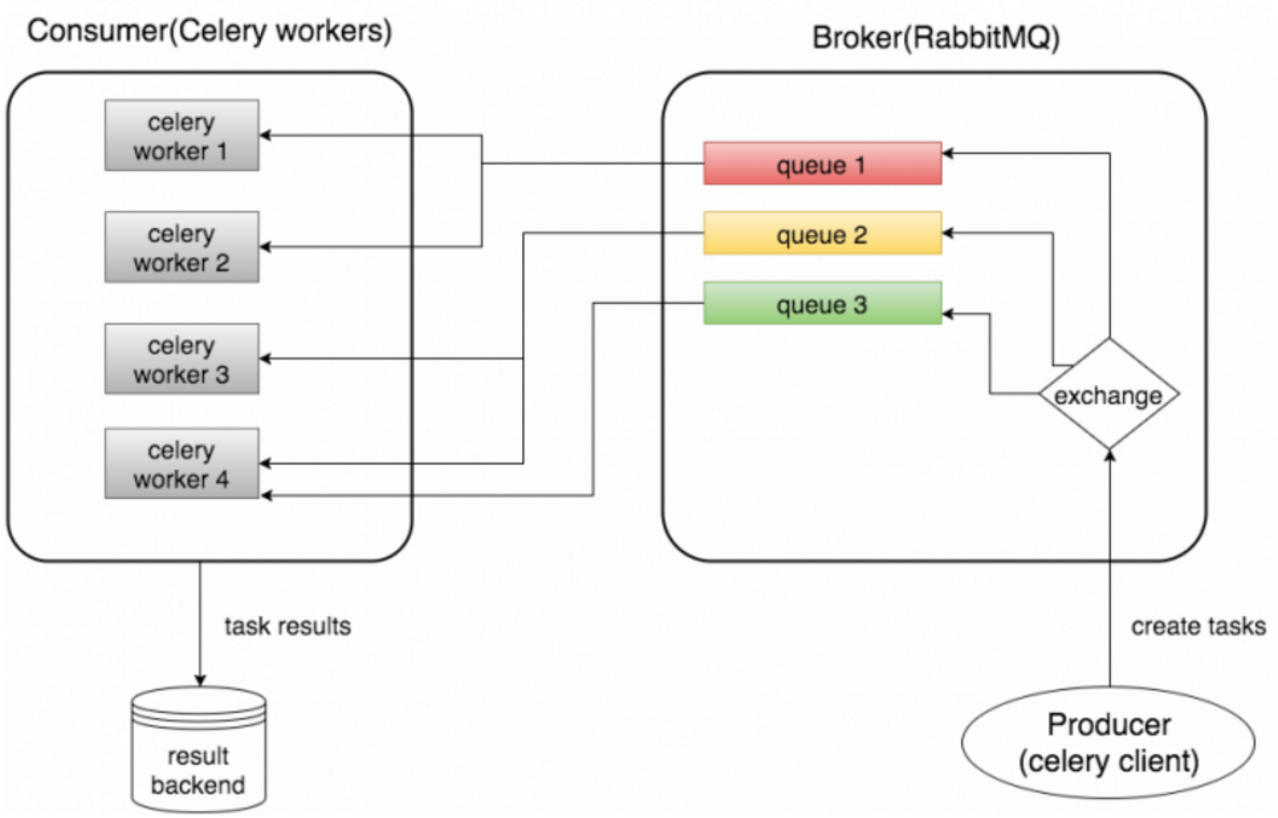
Celery and RabbitMQ

Celery is the default task queue for GeoNode. It provides:

- asynchronous task queue/job queue based on distributed message passing
- focused on real-time operation, but supports scheduling as well
- the execution units, called tasks, are executed concurrently on a single or more worker servers
- it supports many message brokers (RabbitMQ, Redis, MongoDB, CouchDB, ...)
- written in Python but it can operate with other languages
- great integration with Django (it was born as a Django application)
- great monitoring tools (Flower, django-celery-results)

RabbitMQ is the messabe broker which comes by default with GeoNode (it is possible to replace it with something different like Redis):

- as a message broker: it accepts and forwards messages
- it is by far the most widely deployed open source broker (35k+ deployments)
- support many message protocols
- supported by many operating systems and languages



Using Celery in your application

In this tutorial you will improve the application developed in the previous tutorial. Previously you had a script which was syncing all of the GeoNode layers metadata to Solr (*foss4g_scripts/geonode2solr*). Instead than running this script periodically for all of the layers, a better approach would be to syncronize in Solr a layer as soon as the metadata for that layer are saved in GeoNode by the user.

As the metadata Solr synchronization process can slow down the user interaction with the application, you will send that process asynchronously using Celery and RabbitMQ. You will see how you can use Celery to reliably process these tasks, and how to use the Flower Celering monitoring tool to analyze the processed tasks.

Using Django signals

You need a way to run the synchronization process to Solr from GeoNode as soon as the user save the metadata for a given layer.

One way to accomplish this is by forking the GeoNode metadata update view. But forking is the wrong way to do things as it introduces a lot of complications when updating to a newer GeoNode version.

So, how can you execute the sync process without forking GeoNode? Django signals provides a convenient way to do this.

> Django includes a “signal dispatcher” which helps allow decoupled applications get notified when actions occur elsewhere in the framework. In a nutshell, signals allow certain senders to notify a set of receivers that some action has taken place. They’re especially useful when many pieces of code may be interested in the same events.

Django provides a set of built-in signals that let user code get notified by Django itself of certain actions. These include some useful notifications like for example the notification that happens when saving an instance.

You will use the post_save signal which is run when a GeoNode layer is saved to run the Solr synchronization code.

For this purpose, create an *geonode/solr/utils.py* file and copy this python code, readapted from a previous tutorial (check the *foss4g_scripts/geonode2solr.py* file):

```
import json
import requests
```

```
    def layer2dict(layer):
        """
        Return a json representation for a GeoNode layer.
        """
        category = ''
        if layer.category:
            category = layer.category.gn_description
        wkt = "ENVELOPE(%s,%s,%s,%s)" % (layer.bbox_x0, layer.bbox_x1, layer.bbox_y1, lay
        layer_dict = {
                        'id': str(layer.uuid),
                        'name': layer.name,
                        'title': layer.title,
                        'abstract': layer.abstract,
                        'bbox': wkt,
                        'category': category,
                        'modified_date': layer.date.isoformat()[0:22] + 'Z',
                        'username': layer.owner.username,
                        'keywords': [kw.name for kw in layer.keywords.all()],
                        'regions': [region.name for region in layer.regions.all()],
                    }
        print layer_dict
        return layer_dict


    def layer_to_solr(layer):
        """
        Sync a layer in Solr.
        """

        layer_dict = layer2dict(layer)

        layer_json = json.dumps(layer_dict)

        url_solr_update = 'http://localhost:8983/solr/boston/update/json/docs'
        headers = {"content-type": "application/json"}
        params = {"commitWithin": 1500}
        res = requests.post(url_solr_update, data=layer_json, params=params,  headers=hea
        print res.json()
```

Now, create the *geonode/solr/signals.py* file and add this code in it:

```
from django.db.models.signals import post_save
from geonode.layers.models import Layer
from .utils import layer_to_solr

def sync_solr(sender, instance, created, **kwargs):
    print 'Syncing layer %s with Solr' % instance.typename
    layer_to_solr(instance)
```

```
post_save.connect(sync_solr, sender=Layer)
```

Thanks to the post_save signal, sync_solr will be run every time a layer is saved.

Finally import signals in *geonode/solr/**init**.py*:

```
import signals
```

Test the signal

Now to test the layer post_save signal you just created, try updating one of the layer (for this purpose, go the layer page, then click on *Edit Layer* then on *Edit Metadata*)

Change some of the metadata and then check if in Solr the metadata you updated were correctly synced:

http://localhost:8983/solr/boston/select?indent=on&q=name:%22biketrails_arc_p%22&wt=json

(you need to change the q=name parameter to your layer's name)

Process asynchronously with Celery

Now you found a great way to run the *layer_to_solr* method without forking GeoNode, but you are still doing a synchronous processing. Let's add Celery and RabbitMQ to the mix!

Add this at the end of *geonode/local_settings.py*:

```
BROKER_URL = 'amqp://guest:guest@localhost:5672//'
CELERY_ALWAYS_EAGER = False
```

BROKER_URL is the location where RabbitMQ is running. You will send the tasks as the *guest* user. In production it is recommendable to create a specific user with a strong password.

Now create this *geonode/solr/celery_app.py* file, which will make your custom task discoverable by Celery:

```
from __future__ import absolute_import
import os
from celery import Celery
```

```
from django.conf import settings

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'geonode.settings')
app = Celery('geonode')
app.config_from_object('django.conf:settings')
apps = settings.INSTALLED_APPS + ('geonode.solr',)
app.autodiscover_tasks(lambda: apps)
```

Create this *geonode/solr/tasks.py* file which contain the Celery task (the **consumer**, in the jargon, while the post_save signal is the **producer**):

```
from __future__ import absolute_import

from geonode.layers.models import Layer
from celery import shared_task
from .utils import layer_to_solr


@shared_task
def sync_to_solr(layer_id):
    layer = Layer.objects.get(pk=layer_id)
    print 'Syncing layer %s with Solr' % layer.typename
    layer_to_solr(layer)
```

Now modify in this way the *geonode/solr/signals.py* file:

```
from django.db.models.signals import post_save
from geonode.layers.models import Layer
#from .utils import layer_to_solr
from .tasks import sync_to_solr


def sync_solr(sender, instance, created, **kwargs):
    # layer_to_solr(instance)
    sync_to_solr.delay(instance.id)


post_save.connect(sync_solr, sender=Layer)
```

Let's see if everything works. Run the Celery worker by opening another shell (in production you should consider using a tool such as supervisord):

```
$ . /workshop/env/bin/activate
$ cd /workshop/geonode/geonode/solr
$ celery -A celery_app worker -l info
```

Try saving the metadata of a layer. Looking at the Celery log you should see that the Solr task is being executed:

```
$ celery -A celery_app worker -l info
 -------------- celery@ubuntu-xenial v3.1.25 (Cipater)
---- **** -----
--- * ***  * -- Linux-4.4.0-89-generic-x86_64-with-Ubuntu-16.04-xenial
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         geonode:0x7f85b2b79bd0
- ** ---------- .> transport:   amqp://guest:**@localhost:5672//
- ** ---------- .> results:     disabled://
- *** --- * --- .> concurrency: 2 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> default          exchange=default(direct) key=default


[tasks]
  . geonode.services.tasks.harvest_service_layers
  . geonode.services.tasks.import_service
  . geonode.solr.tasks.sync_to_solr
  . geonode.tasks.deletion.delete_layer
  . geonode.tasks.deletion.delete_map
  . geonode.tasks.deletion.delete_orphaned_document_files
  . geonode.tasks.deletion.delete_orphaned_thumbs
  . geonode.tasks.email.send_email
  . geonode.tasks.email.send_queued_notifications
  . geonode.tasks.update.create_document_thumbnail
  . geonode.tasks.update.geoserver_update_layers

[2017-08-07 16:39:32,161: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:56
[2017-08-07 16:39:32,174: INFO/MainProcess] mingle: searching for neighbors
[2017-08-07 16:39:33,181: INFO/MainProcess] mingle: all alone
[2017-08-07 16:39:33,194: WARNING/MainProcess] celery@ubuntu-xenial ready.
[2017-08-07 16:40:11,846: INFO/MainProcess] Received task: geonode.solr.tasks.sync_to
[2017-08-07 16:40:11,910: WARNING/Worker-2] Syncing layer geonode:shape_1 with Solr
[2017-08-07 16:40:11,939: WARNING/Worker-2] {'category': u'Farming', 'modified_date':
[2017-08-07 16:40:11,993: WARNING/Worker-2] {u'responseHeader': {u'status': 0, u'QTim
[2017-08-07 16:40:11,994: INFO/MainProcess] Task geonode.solr.tasks.sync_to_solr[6ca1
```

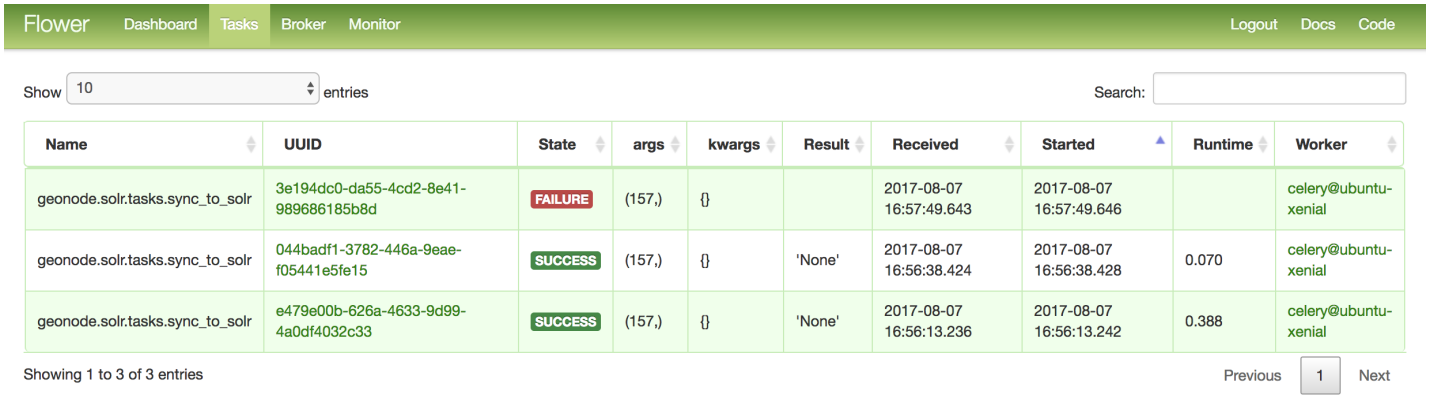
Celery Monitoring

A great Celery monitoring tool is Flower.

Install and run Flower at port 5555:

```
$ . /workshop/env/bin/activate
$ cd /workshop/geonode/geonode/solr/
$ pip install flower==0.9.2
$ celery flower -A celery_app --port=5555
```

Now browse at http://localhost:5555 and you should see the Flower main interface. Try saving some of the layers, and you should see a new task for each layer you save in Flower.



If you want to see a failing task, as in the previous image, one way is to stop Solr and then saving a layer. You can see details of the error which made the task failing by clicking on the failing task itself in the Flower interface