Minerva Documentation

Release 0.1.0

Kitware

July 13, 2017

Contents

CHAPTER 1

What is Minerva?

Minerva is an open source platform for geospatial visualization using Girder as its datastore.

Currently Minerva is pegged to Girder version 971d7b60710dba77da626cdbb0078bb78c951bbb. To update the version of Girder that Minerva depends on, update the version in the *.girder-version* file in Minerva's root directory.

CHAPTER 2

Glossary

Source

Sources are likely going to be removed, at least from the UI. The intention behind a source is to produce data, i.e. to produce a specific dataset, and to describe How/What/Why of data. A source also tracks provenance information. The current plan is to remove sources from the UI, and to have the functionality that Sources provide appear in the Datasets panel. All of the source info could be contained in a *source* or *source_type* key in the minerva metadata.

A source produces data. A source itself cannot be visualized, but a source can create a dataset that can be visualized, or it can be the input to an analysis, which then creates a dataset that can be visualized.

See also:

Information on how to create a Minerva source can be found in *Creating a Source*.

Dataset

A dataset contains data, and it may be possible to visualize a dataset.

Currently Minerva supports visualizing a dataset only on a map.

A dataset is a Girder item that lives in the User's Minerva/Dataset folder and has *minerva* namespaced metadata.

Currently the metadata keys for a dataset are a jumbled confusion, and should be reworked for clarity.

Dataset Minerva Metadata

This is not exhaustive, but provides some coverage. Also describes problems with the current approach.

dataset_type

This has some notion about the type/format of data, but also includes some info about how the dataset should be rendered on a map. There should be some consistent API based around this.

Type/Format of a dataset is independent of the source, e.g. a json file stored in a Girder Item, a mongo collection, and a call to many APIs can produce results of json or table/json.

json: The first file in the item likely had a .json extension.

wms: A wms layer imported from a wms source (server). Probably this should not be a dataset because we have no access to the underlying data.

geojson: A dataset that contains a geojson file or a geojson file derived from some other file that is in the Dataset. Ideally this could be rendered by GeoJs passing it to the 'json' reader, but it isn't so simple. Choropleth geojson needs to parse and postprocess the geojson specially. If a set of points was displayed but we wanted to specify clustering properties, we would also need to do that using specific rendering properties, and couldn't just pass it to the 'json' reader.

csv: A dataset that contains tabular, csv data.

source_type A partial attempt at provenance tracking and a partial implementation of a replacement for Source as a domain model. 'item', 'mmwr_data_import', 'bsve_search' are examples.

original_type The type of file originally uploaded to the Item. A half-hearted attempt at provenance tracking. Used for datasets that have json as the output of some analysis, but where we already know the structure of the json and can immediately convert it to some renderable type (usually geojson) using a known mapping.

geojson_file 'name' and '_id' of a geojson file, if the Dataset has a geojson file.

csv_preview A preview of a small subset of csv rows from the head of the file.

json_row An example row of a json array. Can be used to present a mapping UI to the user, so they can select examine and select properties or trigger a conversion.

mapper Storage for mapping values, currently used for json to geojson conversion, keeping the latitudeKeypath and longitudeKeypath, both expressed in JSONSchema format.

original_files Array of '_id' and 'name' of files originally uploaded to the Item.

geo_render Contains 'type' of GeoJs rendering, among ('choropleth', 'contour', 'geojson', 'wms'). Also 'file_id' pointing to file data in Girder, if required by the rendering *type*.

geojson Contains the geojson data directly in the metadata under this key.

Analysis

An analysis creates a dataset, but running some client side or server side process, and potentially using datasets and sources as inputs.

Session

CHAPTER 3

Table of Contents

Installation

These installation instructions are aimed at developers and will install Girder, and Minerva from source.

The top level directory of Girder cloned by git will be GIRDER_DIR.

Install of system dependencies

Ubuntu 14.04

Update apt package indices before you start.

```
sudo apt-get update
```

- See Girder system prerequisites for Ubuntu
- See Girder install from source
- Install Minerva system dependencies

```
sudo apt-get install libgdal-dev libnetcdf-dev libpng12-dev pkg-config
```

Fedora 22

```
sudo dnf install git gcc-c++ libffi-devel make python-devel python-pip geos-devel gdal-devel netcdf-
```

- See installing mongo on Red Hat
- See installing node.js on Red Hat

Setup Girder admin user and assetstore

- Run Girder to ensure that it works. Mongo should already be running, and you should follow the instructions for a source tree install.
- Navigate to Girder in your browser, register an admin user.
- Navigate to the Admin console in Girder, when you are logged in as an admin user, then click on the Assetstores section.

- Create a default Assetstore.

Install database_assetstore as a Girder plugin

This is needed because minerva depends on database_assetstore plugin.

- Install girder_db_items plugin into the Girder plugins directory.

```
cd GIRDER_DIR/plugins
git clone https://github.com/OpenGeoscience/girder_db_items database_assetstore

- Install the required python packages for the database_assetstore plugin.

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

Install of Minerva as a Girder plugin

- Install Minerva into the Girder plugins dir from source.

```
cd GIRDER_DIR/plugins
git clone https://github.com/Kitware/minerva.git
```

- Install the Python dependencies of Girder plugins, including dev dependencies.

```
cd GIRDER_DIR
export IGNORE_PLUGINS=celery_jobs,geospatial,google_analytics,hdfs_assetstore,jquery_widgets,metadata
scripts/InstallPythonRequirements.py --mode=dev --ignore-plugins=${IGNORE_PLUGINS}
```

Notes:

- If the cryptography pip package in Girder fails to install, or fails when running Girder, try this

```
sudo pip uninstall cffi
sudo pip uninstall cryptography
sudo pip install -U cryptography
```

- Build the client side of Minerva

```
cd GIRDER_DIR
npm install
```

- copy the `minerva.dist.cfg` file, located in the GIRDER_DIR/plugins/minerva/server/conf directory, to `minerva.local.cfg` in that same directory. Any property in `minerva.local.cfg` will take precedent over any property with the same name in `minerva.dist.cfg`. If the `minerva.local.cfg` file is absent, values will be read from `minerva.dist.cfg`. Change the `encrypt_key` value in `minerva.local.cfg` file; the value should be a 32 byte url-safe base-64 encoded string. You can either replace the existing string with one of equal length, using letters and numbers, and ending with an '=', or generate one within python with the following code

```
from cryptography.fernet import Fernet
Fernet.generate_key()
```

- Run the Girder server

```
cd GIRDER_DIR
python -m girder
```

- Navigate to the Admin console in Girder, when you are logged in as an admin user, then click on the Plugins section.
- Enable the Minerva plugin, which will enable Gravatar, and Jobs plugins. Click the button to restart the server.

This will serve Minerva as your top level application. Girder will now be served at your top level path with `/girder`.

- When the server is restarted, refresh the page, you will need to remove #/plugins from your URL as this is no longer valid.

Example:

Pre-Minerva:

http://localhost:8080 => serves Girder

Post-Minerva:

http://localhost:8080 => serves Minerva

http://localhost:8080/girder => serves Girder

Data services

Several minerva components rely on having a data services server up and running. You can either connect to an existing server or spin up a local server using vagrant. See the *Provisioning data services* section for more details.

Provisioning data services

Minerva uses ansible playbooks to provision servers for its data services. Along with vagrant these playbooks provide a repeatable script to spin up a new data services instance in a few simple commands. Once you have vagrant and ansible installed start up a new data services server as follows.:

```
cd data_services/deploy/ansible
ansible-galaxy install -r requirements.txt
vagrant up
```

This will download a fresh image of Ubuntu server from vagrant's online repository and procede to execute the playbook inside the virtual machine. Once this process is complete you should have several fully functional services running locally.

1. geonode: http://192.168.33.12 user: `admin` password: `geonode`
2. geoserver: http://192.168.33.12:8080/geoserver user: `admin` password `geoserver`
3. flower: http://192.168.33.12:5555

The `celery` tasks exposed in flower come from dataqs and will run periodically to ingest new data from public data sets. The new data will show up as layers within geonode.

Developer Documentation

Installation

These installation instructions are aimed at developers and will install Girder, and Minerva from source.

The top level directory of Girder cloned by git will be GIRDER_DIR.

Install of system dependencies

Ubuntu 14.04

Update apt package indices before you start.

```
sudo apt-get update
```

- See Girder system prerequisites for Ubuntu
- See Girder install from source
- Install Minerva system dependencies

```
sudo apt-get install libgdal-dev libnetcdf-dev libpng12-dev pkg-config
```

Fedora 22

```
sudo dnf install git gcc-c++ libffi-devel make python-devel python-pip geos-devel gdal-devel netcdf-c
```

- See installing mongo on Red Hat
- See installing node.js on Red Hat

Setup Girder admin user and assetstore

- Run Girder to ensure that it works. Mongo should already be running, and you should follow the instructions for a source tree install.
- Navigate to Girder in your browser, register an admin user.
- Navigate to the Admin console in Girder, when you are logged in as an admin user, then click on the Assetstores section.
- Create a default Assetstore.

Install database_assetstore as a Girder plugin

This is needed because minerva depends on database_assetstore plugin.

- Install girder_db_items plugin into the Girder plugins directory.

```
cd GIRDER_DIR/plugins
git clone https://github.com/OpenGeoscience/girder_db_items database_assetstore

- Install the required python packages for the database_assetstore plugin.

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

Install of Minerva as a Girder plugin

- Install Minerva into the Girder plugins dir from source.

```
cd GIRDER_DIR/plugins
git clone https://github.com/Kitware/minerva.git
```

- Install the Python dependencies of Girder plugins, including dev dependencies.

```
cd GIRDER_DIR
export IGNORE_PLUGINS=celery_jobs,geospatial,google_analytics,hdfs_assetstore,jquery_widgets,metadata
scripts/InstallPythonRequirements.py --mode=dev --ignore-plugins=${IGNORE_PLUGINS}
```

Notes:

- If the cryptography pip package in Girder fails to install, or fails when running Girder, try this

```
sudo pip uninstall cffi
sudo pip uninstall cryptography
sudo pip install -U cryptography
```

- Build the client side of Minerva

```
cd GIRDER_DIR
npm install
```

- copy the `minerva.dist.cfg` file, located in the GIRDER_DIR/plugins/minerva/server/conf directory, to `minerva.local.cfg` in that same directory. Any property in `minerva.local.cfg` will take precedent over any property with the same name in `minerva.dist.cfg`. If the `minerva.local.cfg` file is absent, values will be read from `minerva.dist.cfg`. Change the `encrypt_key` value in `minerva.local.cfg` file; the value should be a 32 byte url-safe base-64 encoded string. You can either replace the existing string with one of equal length, using letters and numbers, and ending with an '=', or generate one within python with the following code

```
from cryptography.fernet import Fernet
Fernet.generate_key()
```

- Run the Girder server

```
cd GIRDER_DIR
python -m girder
```

- Navigate to the Admin console in Girder, when you are logged in as an admin user, then click on the Plugins section.
- Enable the Minerva plugin, which will enable Gravatar, and Jobs plugins. Click the button to restart the server.

This will serve Minerva as your top level application. Girder will now be served at your top level path with `/girder`.

- When the server is restarted, refresh the page, you will need to remove #/plugins from your URL as this is no longer valid.

Example:

Pre-Minerva:

> http://localhost:8080 => serves Girder

Post-Minerva:

> http://localhost:8080 => serves Minerva
>
> http://localhost:8080/girder => serves Girder

Data services

Several minerva components rely on having a data services server up and running. You can either connect to an existing server or spin up a local server using vagrant. See the *Provisioning data services* section for more details.

API Documentation

Extending Minerva

Creating a Minerva Plugin

Minerva plugins are **identical** to Girder plugins with the exception that they have a hard dependency on Minerva. This ensures that Minerva will be loaded before your plugin is.

Minerva utilizes the Girder plugin system, so it will be worthwhile to familiarize yourself with their section on Plugin Development.

Below is an example configuration for a Minerva plugin, note the dependency on Minerva:

```
{
    "name": "My Minerva Plugin",
    "dependencies": ["minerva"]
}
```

Your new Minerva plugin should provide no new functionality at this point, and can be enabled through the administration console.

Extending the look and feel of Minerva

While Minerva plugins follow the guidelines provided by Girder's client side, there are additional concerns when writing for Minerva.

Note: Girder has documentation on extending the client-side application which also applies to Minerva.

Minerva Panel Views

Minerva renders panels in a particular way, as a result there are some guidelines that need be followed when creating your own Panels:

- Panels should never call render within their initialize function
- Panel views must extend the base Panel view

```
minerva.views.CoolNewPanel = minerva.views.Panel.extend({
```

- Panel views need to call their parent initialize method to take advantage of collapsible, removable, configurable panels

```
// inside CoolNewPanel's initialize function
minerva.views.Panel.prototype.initialize.apply(this);
```

Configure Minerva's layout

Taking our example plugin from before, we can alter how Minerva displays different panels.

Let's pretend our use case of Minerva deems the Jobs Panel useless, and is more focused on the datasets available. In this case one might want to disable the Jobs Panel entirely, and move the datasets panel to the top.

From your plugin root, create a JavaScript file at `web_client/js/some-file.js` and add the following code:

```
girder.events.once('m:pre-render-panel-groups', function (sessionView) {
    var leftPanelGroup = sessionView.getPanelGroup('m-left-panel-group');

    // Disable/remove the jobs panel
    sessionView.disablePanel('m-jobs-panel');

    // Move the 'Available Datasets' panel to the top
    leftPanelGroup.panelViews.sort(function (a, b) {
        if (a.id === 'm-data-panel') {
            return -1;
        } else if (b.id === 'm-data-panel') {
            return 1;
        } else {
            return 0;
        }
    });
});
```

Above we utilize the `m:pre-render-panel-groups` event to hook into Minerva before any panels are actually rendered, this gives full control over what the final layout looks like.

Creating a Source

I think it's easier to work on the backend first, as you can test the api independently of the client. We'll work through an example using the Elasticsearch source.

Source API

Create an endpoint to create the source, copy server/rest/wms_source.py to server/rest/elasticsearch_source.py and modify accordingly.

Important points are - use the access decorator to ensure only logged in users can call the endpoint - create minerva_metadata with the correct source_type - save the source using the superclass method createSource - return the document corresponding to the new source - here we store the authentication credentials after encryption - set the description object to display the params correctly on the swagger api page

Add the endpoint to server/loader.py

```
info['apiRoot'].minerva_source_elasticsearch = elasticsearch_source.ElasticsearchSource()
```

You should now be able to see your endpoint through the swagger api page, and test it there. Usually

```
http://localhost:8080/api
```

Testing the Source API

Create a test, copy plugin_tests/wms_test.py to plugin_tests/elasticsearch_test.py.

Add the test to plugin.cmake

```
add_python_test(elasticsearch PLUGIN minerva)
```

Run cmake PATH_TO_GIRDER_DIR again in your build directory to pick up the new test.

Now you should see the new test in your build directory

```
ctest -N | grep minerva
  Test #110: server_minerva.dataset
  Test #111: server_minerva.source
  Test #112: server_minerva.session
  Test #113: server_minerva.analysis
  Test #116: server_minerva.import_analyses
  Test #117: server_minerva.contour_analysis
  Test #118: server_minerva.wms
  Test #119: server_minerva.elasticsearch
  Test #120: server_minerva.geojson
  Test #121: server_minerva.mean_contour_analysis
  Test #122: pep8_style_minerva_constants
  Test #124: pep8_style_minerva_rest
  Test #125: pep8_style_minerva_utility
  Test #126: pep8_style_minerva_bsve
  Test #127: pep8_style_minerva_jobs
  Test #128: jshint_minerva
  Test #129: jsstyle_minerva
  Test #130: web_client_minerva
```

You can run the test, with extra verbosity

```
ctest -R server_minerva.elasticsearch -VV
```

Also check your python style, and fix any errors

```
ctest -R pep8_style_minerva_rest -VV
```

Add the source to client side collection

Add the new source type to web/external/js/collections/SourceCollection.js, this will prevent mysterious backbone errors later on like

```
a.on is not a function
```

You're welcome.

Add a new Source Model

Add a new model like web_external/js/models/ElasticsearchSourceModel.js.

Add the Source to AddSourceWidget

Create a widget to add your new source type, e.g. AddElasticsearchSourceWidget

```
web_external/js/views/widgets/AddElasticsearchSourceWidget.js
web_external/templates/widgets/addElasticsearchSourceWidget.jade
```

Add your new source type along with a reference to the create widget to the dictionary in AddSourceWidget.js.

Test that when you click on the add new source icon in the source panel, your new source type is displayed as an option.

Add an Action to the Source Displayed in the Source Panel

If it makes sense for your source to have an action, as when there is a natural path to create a dataset from your source, add an action to the source displayed in the source panel.

E.g., a WMS source naturally creates datasets by exposing a set of WMS layers and allowing one or more to be created as a dataset. An Elasticsearch source naturally creates datasets by running an analysis which is a search query, resulting in a JSON dataset with a default visualization as GeoJson.

Add an event handler for your source icon in web_external/js/views/body/SourcePanel.js .

Add the widget constructed and rendered by the event handler

```
web_external/js/views/widgets/ElasticsearchWidget.js
web_external/templates/widgets/elasticsearchWidget.jade
```

Comply with Javascript styles

Because it's the law of the land.

```
ctest -R jshint_minerva -VV
ctest -R jsstyle_minerva -VV
```

User Documentation

Time Series

Data structure

The dataset should in JSON format, the root level should be a JSON array. It would contain an arbitrary number of frame objects. Each `frame` object should have two required properties, `geojson` and `time`, and one optional `label` property.

The `time` property should have a value of ISO-8601 DateTime string to indicate the time for this frame. However, instead of the time property, the order in the JSON array defines the order of frames.

If the `label` property is specified, the value of it will be displayed on the layer control under *Current Display*, otherwise, the value of the `time` property will be displayed.

The `geojson` property should have a value of a valid GeoJSON object, e.g. Point, Polygon, Feature, or FeatureCollection. If GeoJSON Feature has been used, styles defined inside the properties object will be used to style the geometry.

An example minimal Time Series dataset contains one frame and one styled GeoJSON feature point looks like below

```
[
  {
    "geojson": {
      "type": "Feature",
      "geometry": {
```

```
        "type": "Point",
        "coordinates": [
          -118.87264913546323,
          36.69629877019643
        ]
      },
      "properties": {
        "annotationType": "point",
        "strokeWidth": 3,
        "name": "Camp",
        "fillColor": "#00ff00",
        "annotationId": 2,
        "strokeOpacity": 1,
        "scaled": false,
        "stroke": true,
        "radius": 10,
        "fillOpacity": 0.25,
        "strokeColor": "#000000",
        "fill": true
      }
    },
    "time": "2017-02-24T15:07:14.188798Z"
  }
]
```


Frame controls

Once a Time Series dataset has been uploaded. It can be added as a layer to the layers panel. In addition to standard layer controls, Time Series layer has a special set of controls. There will be a frame slider, a play button, a stop button, a previous frame button, a next frame button, and a `duration` selection.

Sliding the Frame slider will make different frames of the Time Series visible. Only the selected frame will be visible on the map.

Clicking the play button will start to animate each frame in the dataset. At this moment, each frame will get the equal visible time. The time that each frame will be visible equals the `duration` divided by the total number of frames.

When the Time Series is being animated, a pause button will be available to pause the animation. Clicking the stop button will set the selected frame to the first frame in the Time Series.

Clicking the previous frame button will set the frame to the previous frame of the currently selected frame. If not available, the last frame will be used. Clicking the next frame button will set the frame to the next frame of the currently selected frame. If not available, the first frame will be used.

Changing the `duration` with the duration selection will change the time it will take the whole series to animate from the first frame to the last frame.

Geocoder

Installation

Minerva Geocoder requires Twofishes to be up and running. We need Vagrant and Ansible for this purpose.

Install ansible

```
$ pip install ansible
```

Clone Twofishes

```
$ git clone git@github.com:OpenGeoscience/fsqio.git
$ cd fsqio/ansible
```

Install requirements

```
$ ansible-galaxy install -r requirements.txt
$ vagrant up geocoder
```

This operation takes some time. Once the provisioning is successful you should see Twofishes running on http://localhost:8087.

Usage

You can use Swagger to interact with the geocoder. Currently there are 3 endpoints.

1. **Autocomplete**

   Autocompletes a given string and returns 10 matching location names. Following example shows how the string "bos" gets autocompleted.

   Parameters:

   twofishes : http://localhost:8087

   location : bos

   Response:


```
[
  "Boston, MA, United States",
  "Boshan, Shandong, China",
  "Bosaso, Bari, Somalia",
  "Bossier City, LA, United States",
  "Boston, Lincolnshire, United Kingdom",
  "Boscoreale, Campania, Italy",
  "Bossangoa, Central African Republic",
  "Boshkengash, Tajikistan",
  "Boskoop, Netherlands",
  "Bosanska Krupa, Bosnia and Herzegovina"
]
```


2. **Getting Geojson String**

   Gets a Geojson response for a given place name/names as a list.

   Parameters:

   twofishes : http://localhost:8087

   locations : ["utah"]

   Response:


```
{
  "features": [
    {
      "geometry": {
        "coordinates": [
          [
            [
```

```
              -113.48151249430609,
              42.0000040881182
            ],
            [
              -113.29450077529052,
              42.0000040881182
            ],
            [
              -113.10748905627487,
              42.0000040881182
            ],
            [
              -112.92047733815859,
              42.0000040881182
            ]
          ]
        ],
        "type": "Polygon"
      },
      "properties": {
        "location": "utah"
      },
      "type": "Feature"
    }
  ],
  "type": "FeatureCollection"
}
```


3. **Create Minerva Dataset**

   Creates a minerva dataset from a Twofishes search result. Each geojson feature will include the property "location" which will inlude the name.

   Parameters:

   twofishes : http://localhost:8087

   location : ["san francisco", "boston"]

   name : some_cities.geojson

   Response:

   Similar to this response. There will be a some_cities.geojson dataset in minerva which you can plot on the map.


```
{
  "_id": "59396a6778e55a3f9c8fbecb",
  "assetstoreId": "5900f48778e55a10051679aa",
  "created": "2017-06-08T15:16:55.042802+00:00",
  "creatorId": "58f912e478e55a2da26776e5",
  "exts": [
    "geojson"
  ],
  "itemId": "59396a6778e55a3f9c8fbec9",
  "mimeType": "application/octet-stream",
  "name": "some_cities.geojson",
  "path": "87/b3/87b3b3d077c4ce4e16477a1c2f352f1a9f6607c979c090d75257122cbd085837c76cf55d78f1b3f7d3e8
  "sha512": "87b3b3d077c4ce4e16477a1c2f352f1a9f6607c979c090d75257122cbd085837c76cf55d78f1b3f7d3e8d36b
  "size": 132745
}
```

API Documentation