GFZ
Helmholtz Centre
POTSDAM

Routing-WS Documentation

Release 1.1.1

Javier Quinteros

November 10, 2017

CONTENTS

1 Summary **1**

2 Installation **3**
- 2.1 License . . . 3
- 2.2 Requirements . . . 3
- 2.3 Download . . . 3
- 2.4 Installation on Apache . . . 3
- 2.5 Testing the service . . . 7
- 2.6 Maintenance . . . 8
- 2.7 Upgrade . . . 8

3 Using the Service **9**
- 3.1 Default configuration . . . 9
- 3.2 Manual configuration . . . 9
- 3.3 Importing remote routes . . . 10
- 3.4 Cache of station names and locations . . . 10
- 3.5 Defining virtual networks . . . 11
- 3.6 Methods available . . . 11

4 Documentation for developers **17**
- 4.1 Routing module . . . 17
- 4.2 Utils module . . . 17
- 4.3 Wsgicomm module . . . 22

Python Module Index **25**

Index **27**

CHAPTER ONE

SUMMARY

One of the aims of the European Integrated Data Archive (EIDA) is to provide transparent access and services to high quality, seismic data across different data archives in Europe. In the context of the design of the *EIDA New Generation* (EIDA-NG) software we envision a future in which many different data centers offer data products using compatible types of services, but pertaining to different seismic objects, such as waveforms, inventory, or event data. EIDA provides one example, in which data centers (the EIDA "nodes") have long offered Arclink and Seedlink services, and now offer FDSN web services, for accessing their holdings. In keeping with the distributed nature of EIDA, these services could run at different nodes. Depending on the type of service, these may only provide information about a reduced subset of all the available waveforms.

To assist users to locate data, we have designed a Routing Service, which could run at EIDA nodes or elsewhere, including on a user's personal computer. This (meta)service is supposed to be queried by clients (or other services) in order to localize the address(es) where the desired information is provided.

The Routing Service must serve this information in order to help the development of smart clients and/or services of higher level, which can offer the user an integrated view of the whole EIDA, hiding the complexity of its internal structure. However, the Routing Service need not be aware of the extent of the content offered by each service, avoiding the need for a large synchronized database at any place.

The service is intended to be open and able to be queried by anyone without the need of credentials or authentication.

CHAPTER

TWO

INSTALLATION

License

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Requirements

- Python 2.7
- mod_wsgi (if using Apache). Also Python libraries for libxslt and libxml.

Download

Download the tar file / source from the GEOFON web page at http://geofon.gfz-potsdam.de/software.

Note: Nightly builds can be downloaded from Github (https://github.com/EIDA/routing.git).

Untar into a suitable directory visible to the web server, such as */var/www/eidaws/routing/1/*

```
$ cd /var/www/eidaws/routing/1
$ tar xvzf /path/to/tarfile.tgz
```

This location will depend on the location of the root (in the file system) for your web server.

Installation on Apache

To deploy the EIDA Routing Service on an Apache2 web server using *mod_wsgi*:

1. Extract the package in the desired directory. In these instructions we assume this directory is */var/www/eidaws/routing/1/*.
    - If you downloaded the package from the GEOFON website, unpack the files into the chosen directory. (See *Download* above.)
    - If you want to get the package from Github, use the following commands:

```
    $ cd /var/www/eidaws/routing
    $ git clone https://github.com/GEOFON/routing.git 1
    $ cd 1
```

2. Enable *mod_wsgi*. For openSUSE, add 'wsgi' to the list of modules in the APACHE_MODULES variable in */etc/sysconfig/apache2*

   ```
   APACHE_MODULES+=" python wsgi"
   ```

   and restart Apache. You should now see the following line in your configuration (in */etc/apache2/sysconfig.d/loadmodule.conf* for **openSUSE**)

   ```
   LoadModule wsgi_module   /usr/lib64/apache2/mod_wsgi.so
   ```

   You can also look at the output from `a2enmod -l` - you should see wsgi listed.

   For **Ubuntu/Mint**, you can enable the module with the command

   ```
   $ sudo a2enmod wsgi
   ```

   and you can restart apache with:

   ```
   $ sudo service apache2 stop
   $ sudo service apache2 start
   ```

   If the module was added succesfully you should see the following two links in `/etc/apache2/mods-enable`

   ```
   wsgi.conf -> ../mods-available/wsgi.conf
   wsgi.load -> ../mods-available/wsgi.load
   ```

   For any distribution there may be a message like this in Apache's *error_log* file, showing that *mod_wsgi* was loaded

   ```
   [Tue Jul 16 14:24:32 2013] [notice] Apache/2.2.17 (Linux/SUSE)
   PHP/5.3.5 mod_python/3.3.1 Python/2.7 mod_wsgi/3.3 configured
    -- resuming normal operations
   ```

3. Add the following lines to a new file, *conf.d/routing.conf*, or in *default-server.conf*, or in the configuration for your virtual host.

   ```
   WSGIScriptAlias /eidaws/routing/1 /var/www/eidaws/routing/1/routing.wsgi
   <Directory /var/www/eidaws/routing/1/>
       Order allow,deny
       Allow from all
   </Directory>
   ```

   Change */var/www/eidaws/routing/1* to suit your own web server's needs.

4. Change into the root directory of your installation and copy *routing.cfg.sample* to *routing.cfg*, or make a symbolic link

   ```
   $ cd /var/www/eidaws/routing/1
   $ cp routing.cfg.sample routing.cfg
   ```

5. Edit *routing.wsgi* and check that the paths there reflect the ones selected for your installation.

6. Edit *routing.cfg* and be sure to configure everything correctly. This is discussed under "*Configuration Options*" below.

7. Start/restart the web server e.g. as root. In **OpenSUSE**

   ```
   $ /etc/init.d/apache2 configtest
   $ /etc/init.d/apache2 restart
   ```

   or in **Ubuntu/Mint**

```
$ sudo service apache2 reload
$ sudo service apache2 stop
$ sudo service apache2 start
```

8. Get initial metadata in the *data* directory. You have two options to feed the system with some routes. Either you edit by hand (or copy from some other place) a file with your local streams and save them into *data/routing.xml*, or you get them from an Arclink server. In the latter case, you need to allow this in the configuration file like this:

```
$ vim routing.cfg
$ # set ArclinkBased = true to allow the information to be overwritten by Arclink data
$ grep ArclinkBased routing.cfg
ArclinkBased = true
```

   After saving this change you can run change into the *data* directory and run the `updateAll.py` script there.

```
$ cd /var/www/eidaws/routing/1/data
$ ./updateAll.py -l DEBUG
```

   If you don't specify any parameters to the script, the information needed will be read from the configuration file at the default location *../routing.cfg*. You can use the switch *-h* to see the parameters you can use.

```
$ ./updateAlls.py -h
usage: updateAll.py [-h] [-l {CRITICAL,ERROR,WARNING,INFO,DEBUG}] [-s SERVER]
                    [-c CONFIG]

Get EIDA routing configuration and "export" it to the FDSN-WS style.

optional arguments:
  -h, --help            show this help message and exit
  -l {CRITICAL,ERROR,WARNING,INFO,DEBUG}, --loglevel {CRITICAL,ERROR,WARNING,INFO,DEBUG}
                        Verbosity in the output.
  -s SERVER, --server SERVER
                        Arclink server address (address.domain:18001).
  -c CONFIG, --config CONFIG
                        Config file to use.
```

9. It is important to check the permissions of the working directory and the files in it, as some data needs to be saved there. For instance, in some distributions Apache is run by the `www-data` user, which belongs to a group with the same name (`www-data`). The working directory should have read-write permission for the user running Apache **and** the user who will do the regular metadata updates (see crontab configuration in the last point of this instruction list). The system will also try to create and write temporary information in this directory.

   Warning: Wrong configuration in the permissions of the working directory could diminish the performance of the system.

   One possible configuration would be to install the system as a user (for instance, *sysop*), who will run the crontab update, with the working directory writable by the group of the user running Apache (*www-data* in **Ubuntu/Mint**).

```
$ cd /var/www/eidaws/routing/1
$ sudo chown -R sysop.www-data .
$ cd data
$ sudo chmod -R g+w .
```

10. Arrange for regular updates of the metadata in the working directory. Something like the following lines will be needed in your crontab:

```
$ Daily metadata update for routing service
52 03 * * * /var/www/eidaws/routing/1/data/updateAll.py
```

11. Restart the web server to apply all the changes, e.g. as root. In **OpenSUSE**:

```
$ /etc/init.d/apache2 configtest
$ /etc/init.d/apache2 restart
```

or in **Ubuntu/Mint**:

```
$ sudo service apache2 reload
$ sudo service apache2 stop
$ sudo service apache2 start
```

Configuration options

The configuration file contains two sections up to this moment.

Arclink

Warning: The capability to get routes from an Arclink server has been deprecated and it should not be used at all. This functionality will be removed from future versions!

In the Arclink section an arclink server must be defined, from which the default routing table could be retrieved. The default value is the Arclink server running at GEOFON, but this can be configured with the address of any Arclink server.

```
[Arclink]
server = eida.gfz-potsdam.de
port = 18002
```

Service

baseURL should contain the basic URL of the current Routing Service in order to be used in the generation of the *application.wadl* method. For instance,

```
.. code-block:: ini
```

baseURL = http://mydomain.dom/eidaws/routing/1

The variable *info* specifies the string that the `config` method from the service should return.

The variable *updateTime* determines at which moment of the day should be updated all the routing information.

The format for the update time should be `HH:MM` separated by a space. It is not necessary that the different time entries are in order. If no update is required, there should be nothing at the right side of the = character.

Deprecated since version This: functionality was actually skipped and this options will be removed in future releases. The usage of a cronjob is recommended to run the *updateAll.py* script and generate the binary version of the routing table.

ArclinkBased determines whether the routing information should be retrieved from an Arclink server by the *updateAll.py* script. Usually, you want to set it to `false` in order to configure your own set of routes, so that the update procedure will not delete your manual configuration.

Deprecated since version This: option will be removed in a future version, when the capability to feed the service from an Arclink server disappears.

verbosity controls the amount of output send to the logging system depending of the importance of the messages. A number is expected ranging from 1 to 4, meaning: 1: Error, 2: Warning, 3: Info and 4: Debug.

synchronize specifies the remote servers from which more routes should be imported. This is explained in detail in *Importing remote routes*.

allowoverlap determines whether the routes imported from other services can overlap the ones already present. In case this is set to `false` and an overlapping route is found, the Route will be discarded with an error message in the log. When it is set to `true`, the Route will be still included, but the resulting data could be inconsistent.

```
[Service]
baseURL = http://mydomain.dom/eidaws/routing/1
info = Routing information from the Arclink Server at GEOFON.
   All the routes related to EIDA are supposed to be available here.
updateTime = 01:01 16:58
ArclinkBased = true
verbosity = 3
synchronize = SERVER2, http://server2/eidaws/routing/1
    SERVER3, http://server3/eidaws/routing/1
allowoverlap = true
```

Installation problems

Always check your web server log files (e.g. for Apache: `access_log` and `error_log`) for clues.

If you visit http://localhost/eidaws/routing/1/version on your machine you should see the version information of the deployed service

```
1.1.1
```

If this information cannot be retrieved, the installation was not successful. If this **do** show up, check that the information there looks correct.

Testing the service

Two scripts are provided to test the functionality of the service at different levels. These can be found in the `test` folder under the root directory of your installation.

Class level

The script called `testRoute.py` will try to import the objects used in the Routing Service in order to test their functionality. The data will not be provided by the web service, but from the classes inside the package. In this way, the logic of the package and the coherence of the information can be tested, excluding other factors related to the configuration of other pieces of software (f.i. web server, firewall, etc.).

```
$ ./testRoute.py
Running test...
Checking Dataselect CH.LIENZ.*.BHZ... [OK]
Checking Dataselect CH.LIENZ.*.HHZ... [OK]
Checking Dataselect CH.LIENZ.*.?HZ... [OK]
Checking Dataselect GE.*.*.*... [OK]
Checking Dataselect GE.APE.*.*... [OK]
Checking Dataselect RO.BZS.*.BHZ... [OK]
```

A set of test cases have been implemented and the expected responses are compared with the ones returned by the service.

Note: The test cases are related to the sample routing data which is provided in routing.xml.sample and will make no sense if the service is configured to route other set of networks. In that case, the operator of the service should modify scripts in order to test the coherence of the information provided by the service.

Service level

The script called `testService.py` will try to connect to a Routing Service at a particular URL, which **must** be passed as a parameter. In previous versions the default value was http://localhost/eidaws/routing/1/query, but as the functionality of the Routing Service was much improved, real and resolvable addresses are needed.

```
$ ./testService.py http://server/path/query
Running test...
Checking Dataselect CH.LIENZ.*.BHZ... [OK]
Checking Dataselect CH.LIENZ.*.HHZ... [OK]
Checking Dataselect CH.LIENZ.*.?HZ... [OK]
Checking Dataselect GE.*.*.*... [OK]
Checking Dataselect GE.APE.*.*... [OK]
Checking Dataselect GE,RO.*.*.*... [OK]
Checking Dataselect RO.BZS.*.BHZ... [OK]
Checking non-existing network XX... [OK]
Checking incompatibility between alternative=true and format=get... [OK]
Checking the 'application.wadl' method... [OK]
Checking the 'info' method... [OK]
Checking very large URI... [OK]
Checking the 'version' method... [OK]
Checking wrong values in alternative parameter... [OK]
Checking swap start and end time... [OK]
Checking wrong format option... [OK]
Checking unknown parameter... [OK]
```

The set of test cases related to data consistency are the same as in the `testRoute.py` script. The other tests are related to the protocol itself.

Maintenance

The Routing Table needs to be updated regularly due to the small but constant changes in the EIDA structure. You should always be able to run safely the `updateAll.py` script at any time you want. The Routing Service creates a binary version of the XML containing the routes, but this will be automatically updated each time a new inventory XML file is detected.

Upgrade

At this stage, it's better to back up and then remove the old installation first.

```
$ cd /var/www/eidaws/routing/ ; mv 1 1.old
```

Then reinstall from scratch, as in the *installation instructions*. Your web server configuration should need no modification. At Steps 4-6, re-use your previous versions of `routing.wsgi` and `routing.cfg`

```
$ cp ../1.old/routing.wsgi routing.wsgi
$ cp ../1.old/routing.cfg routing.cfg
```

And of course, copy your local routing table also.

```
$ cp ../1.old/data/routing.xml data/routing.xml
$ cp ../1.old/data/masterTable.xml data/masterTable.xml
```

CHAPTER

THREE

USING THE SERVICE

Default configuration

A script called `updateAll.py` is provided in the package, which can be found in the `data` folder. This script can load the local routing information as well as synchronize the remote routes, which are provided by the other EIDA nodes. All necessary parameters will be read from the configuration file (`routing.cfg`). Namely, the list of data centres, which have data to synchronize.

When the service starts, checks if there is a file called `routing.xml` in the `data` directory. This file is expected to contain all the information needed to feed the routing table. The file format must be Arclink-XML.

The following is an example of an Arclink-XML file.

```
<?xml version="1.0" encoding="utf-8"?>
<ns0:routing xmlns:ns0="http://server/ns/Routing/1.0/">
    <ns0:route networkCode="GE" stationCode="" locationCode="" streamCode="">
      <ns0:station address="http://domain/fdsnws/station/1/query"
          priority="1" start="1993-01-01T00:00:00" end="" />
      <ns0:dataselect address="http://domain/fdsnws/dataselect/1/query"
          priority="1" start="1993-01-01T00:00:00" end="" />
    </ns0:route>
    <ns0:route networkCode="CH" stationCode="*" locationCode="*" streamCode="*">
      <ns0:station address="http://domain2/fdsnws/station/1/query"
          priority="1" start="1980-01-01T00:00:00" end="" />
      <ns0:dataselect address="http://domain2/fdsnws/dataselect/1/query"
          priority="1" start="1980-01-01T00:00:00" end="" />
    </ns0:route>
</ns0:routing>
```

This is exactly the file that the `updateAll.py` script creates with information from EIDA. With this information and the metadata downloaded by the same script the service can be started.

Manual configuration

A better option would be to take the file from Arclink as a base and make some adjustments to it manually. The number of routes could be reduced drastically by means of a clever use of the wildcards.

If some extra information not available within EIDA would like to be also routed, there is a *masterTable* that can be used. When the service starts, it checks if a file called `masterTable.xml` in the `data` folder exists. If this is the case, the file is read, the routes inside are loaded in a separate table and are given the maximum priority. This could be perfect to route requests to other data centres, whose internal structure is not well known.

Note: There are two main differences between the information provided in *routing.xml* and the one provided in *masterTable.xml*. The former will be used to synchronize with other data centers if requested. On the other hand, the information added in *masterTable.xml* will be kept private and not take part in any synchronization process.

Warning: Only the network level is used to calculate the routing for the routes in the master table. This makes sense if we consider that the main purpose of this *extra* information is to be able to route requests to other data centres who do **not** synchronize their routing information with you. Therefore, the internal and more specific structure of the distribution of data to levels deeper than the network are usually not known.

In the following example, we show how to point to the service in IRIS, when the `II` network is requested.

```xml
<?xml version="1.0" encoding="utf-8"?>
<ns0:routing xmlns:ns0="http://geofon.gfz-potsdam.de/ns/Routing/1.0/">
    <ns0:route locationCode="" networkCode="II" stationCode="" streamCode="">
        <ns0:dataselect address="service.iris.edu/fdsnws/dataselect/1/query"
            end="" priority="9" start="1980-01-01T00:00:00.0000Z" />
    </ns0:route>
</ns0:routing>
```

Warning: The *priority* attribute will be valid only in the context of the *masterTable*. There is no relation with the priority for a similar route that could be in the normal routing table.

The routes that are part of the `masterTable.xml` will not be sent when the `localconfig` method of the service is called, only the ones in the normal routing table.

The idea is that the routes in the normal routing table is the local information that should be probably synchronized with other Routing Services.

Importing remote routes

In the case case that one datacenter decides to include routes from other datacenter (as in the EIDA case) , there is no need to define them locally.

A normal use case would be that the datacenter *A* needs to provide routing information of data centres *A* **and** *B* to its users. In order to allow datacenter *B* to export its routes, a method called `localconfig` is defined. This method will return to the caller all the routing information locally defined in the `routing.xml` file. Every datacenter is free to restrict the access to this method to well-known IP addresses or to keep it completely open by means of access rules in the web server.

If the datacenter *A* has access to this method, it can import the routes automatically by means of the inclusion of the base URL of the service at datacenter *B* in the *synchronize* option (under *Service*) of its configuration file.

```
[Service]
synchronize = DC-B, http://datacenter-b/path/routing/1
```

When the service in datacenter *A* starts, it will first include all the routes defined in `routing.xml` and then it will save the routes read from http://datacenter-b/path/routing/1/localconfig in a file called `DC-B.xml` under the `data` folder. This file can be used for future reference in case that all the routes need to be updated and datacenter *B* is not available.

Once the file is saved, all the routes inside it will be added to the routing table in memory.

Cache of station names and locations

Once all the routes have been imported, all instances of Station-WS are queried with the available routes in order to get a list of stations and their locations. The main purpose of this is to improve the querying capabilities of the Routing Service in the following two use cases.

- Locations will be used to allow the usage of the FDSN parameters: *minlat*, *maxlat*, *minlon*, and *maxlon*. These can be used to define a rectangular area and **discover** all stations present. This functionality was not possible with the previous version of the Service, as no information related to the streams was saved.

- The case where the station names are needed is quite different. It had been detected that some queries contained the station name (e.g. sta=XYZ) but no information about the network. This could be usual when a user remembers the name of the station, which is associated with a location or village, but does not pay attention to the network to which it belongs. As many routes have a configuration like NETCODE.*.*.*, there was no way to know if the station belonged to that network or not. Meaning that in most of the cases, this resulted in the Routing Service sending the user almost **all** the available routes to further discover where the station is. This type of routes are very practical for the Administrator of the service, but less useful for the user in cases like this. The solution was to create an automatic cache of the station names, so that the Administrator can still configure very short routes, while the user can use more precise filters in his/her request.

Defining virtual networks

The concept of a collection of stations, different of the formal network, has existed since a long time. There are many reasons why this could result interesting. From creating a virtual network code which encompasses many networks, to defining a subset of stations from one or more networks.

Since version 1.1.0 this new feature is supported in a way that the operator can define in the *routing.xml* file a virtual network code as a list of stations.

For instance, an example is shown below where the _VN network is defined as the combination of stations ST1, ST2 from network RN1 and ST3 from network RN2.

```
<?xml version="1.0" encoding="utf-8"?>
<ns0:routing xmlns:ns0="http://geofon.gfz-potsdam.de/ns/Routing/1.0/">
  <ns0:vnetwork networkCode="_VN">
    <ns0:stream networkCode="RN1" stationCode="ST1" locationCode="*" streamCode="*"
      start="2015-01-01T00:00:00" end="2015-12-31T00:00:00" />
    <ns0:stream networkCode="RN1" stationCode="ST2" locationCode="*" streamCode="*"
      start="2015-01-01T00:00:00" end="2015-12-31T00:00:00" />
    <ns0:stream networkCode="RN2" stationCode="ST3" locationCode="*" streamCode="*"
      start="2015-01-01T00:00:00" end="2015-12-31T00:00:00" />
  </ns0:vnetwork>
</ns0:routing>
```

In the case that the synchronization is enabled, the virtual networks will also be synchronized and shared with the other Routing services. They will be treated in the same way as routes. Collisions with remote definitions will be checked and will not be allowed.

The most important point to clarify when using virtual networks in the query is that the Routing Service will **always return real network and stations codes**. In this way, the returned codes can be always used with the downstream services, which most probably will not support virtual network codes, or at least it is not clear that will contain exactly the same definitions of the virtual networks.

Methods available

Description of the service

The `application.wadl` method returns a WADL (web application description layer) description of the interface using the MIME type *application/xml*. Any parameters submitted to the method will be ignored. The WADL describes all parameters supported by this implementation and can be used as an automatic way to determine methods and parameters supported by this service. This information is generated on the fly.

Version of the software

The `version` method returns the implementation version as a simple text string using the MIME type *text/plain*. Any parameters submitted to the method will be ignored. This scheme follows the FDSN web services approach.

The service is versioned according the following three-digit (x.y.z) pattern:

```
SpecMajor.SpecMinor.Implementation
```

where the fields have the following meaning:

1. *SpecMajor*: The major specification version, all implementations sharing this *SpecMajor* value will be backwards compatible with all prior releases. Values are integers starting at 1.
2. *SpecMinor*: The minor specification version, incremented when optional parameters or behavior is added to the previous specification but backwards compatibility is maintained with the previous major versions, i.e. all 1.y.z service versions will be compatible with version 1.0. Values are integers starting at 0.
3. *Implementation*: The implementation version, an integer identifier specific to the data center implementation. Useful to track service updates for bug fixes, etc. but with no implication on conformance to the specification.

Together the *SpecMajor* and *SpecMinor* versions imply a minimum expected behavior of a given service. This versioning scheme allows clients to expect specific behavior based on the *SpecMajor* version, while allowing the extension of the service with optional parameters and maintaining backwards compatibility. Each version number is service specific, there is no implication that *SpecMajor* version numbers across services (from EIDA or FDSN) are related.

Exporting routes

The `localconfig` method reads the content of the `routing.xml` file and returns it when this method is invoked. The MIME type of the returned value is *text/xml*.

See also:

Importing remote routes

Querying information

The `query` method is how the users access the main functionality of the service. Both `GET` and `POST` methods must be supported.

Input parameters

The complete list of input parameters can be seen in *Table 2.1*. Parameter names must be in lowercase, and may be abbreviated as shown, following the FDSN style. Valid input values must have the format shown in the "Format" column. All the values passed as parameters will be case-insensitive strings composed of numbers and letters. No other symbols will be allowed with the exception of:

- wildcards ("`*`" and "`?`"), which may be used to select the streams (for parameters *network*, *station*, *location* and *channel* only), and
- the symbols specified in the ISO 8601 format for dates, namely ':', "–" (minus) and "`.`" may be used for the *starttime* and *endtime* parameters,
- the string "`--`" (two minus symbols) may appear for the location parameter only.

Wildcards are accepted in the case of *network*, *station*, *location* and *channel*. The character `*` matches any value, while `?` matches any character. For any of these parameters, if no value is given it will be set to a star (`*`).

Any of these four parameters may also be submitted as comma-separated lists in order to select two or more values with a single request. For example, the *channel* parameter may be used to specify multiple channels:

```
channel=LHE,LHN,LHZ,BHZ (the individual values may also include wildcards)
```

Blank or empty *location* identifiers may be specified as "--" (two dashes) if needed, which the service must translate to an empty string.

Table 3.1: Input parameters description

Parameter	Support	Format	Description	Default
starttime (start)	Required	ISO 8601	Limit results to time series samples on or after the specified start time.	Any
endtime (end)	Required	ISO 8601	Limit results to time series samples on or before the specified end time.	Any
network (net)	Required	char	Select one network code. This can be either SEED network codes or data center defined codes.	*
station (sta)	Required	char	Select one station code.	*
location (loc)	Required	char	Select one location identifier. As a special case "–" (two dashes) will be translated to an empty string to match blank location IDs.	*
channel (cha)	Required	char	Select one channel code.	*
minlatitude	Required	float	Limit to stations with a	
(minlat)			latitude larger than or equal to the specified minimum.	-90
maxlatitude	Required	float	Limit to stations with a	
(maxlat)			latitude smaller than or equal to the specified maximum.	90
minlongitude	Required	float	Limit to stations with a	
(minlon)			longitude larger than or equal to the specified minimum.	-180
maxlongitude	Required	float	Limit to stations with a	
(maxlon)			longitude smaller than or equal to the specified maximum.	180
service	Required	char	Specify which service will be queried (arclink, seedlink, station, dataselect).	dataselect
format	Required	char	Select the output format. Valid values are: xml, json, get, post	xml
alternative	Optional	boolean	Specify if the alternative routes should be also included in the answer. Accepted values are "true" and "false".	false

Output description and format

There are four different output formats supported by this service. The structure of the information returned is different with each format type. In case of a successful request the HTTP status code will be `200`, and the response will be as described below for each format.

XML format

This is the default selection if the parameter *format* is not specified or if it is given with the value `xml`. The MIME type must be set to *text/xml*. The following is an example of the expected XML structure. Each datacenter element must contain exactly one url element, specifying the URL of the service at a given data centre, exactly one name element, which gives the name of the service a list of params elements, each describing a stream, or set of streams by using appropriate wildcarding, available using the service at that URL. The params element may be repeated as many times as necessary inside the datacenter element.

```
<service>
   <datacenter>
       <url>http://ws.resif.fr/fdsnws/dataselect/1/query</url>
       <params>
           <loc>*</loc>
           <end/>
           <sta>KES28</sta>
           <cha>*</cha>
           <start/>
```

```
            <net>4C</net>
        </params>
        <name>dataselect</name>
    </datacenter>
</service>
```

JSON format

if the format parameter is `json`, the information will be returned with MIME type *text/plain*. The content will be a JSON (JavaScript Object notation) array, in which each element is a JSON object corresponding to a `<datacenter>` element in the XML format shown above. For the example response above, this would appear as:

```
[{"url": "http://ws.resif.fr/fdsnws/dataselect/1/query",
"params": [{"loc": "*", "end": "", "sta": "KES28", "cha": "*", "start": "",
            "net": "4C"}], "name": "dataselect"}]
```

It should be noted that the value associated with params is an array of objects and that there will be as many objects as needed for the same datacenter.

GET format

When the *format* parameter is set to `get`, the output will be declared as *text/plain* and will consist of one URL per line. The URLs will be constructed in a way that they can be used directly by the client to request the necessary information without the need to parse them.

```
http://ws.resif.fr/fdsnws/dataselect/1/query?sta=KES28&net=4C&
    start=2010-01-01T00:00:00&end=2010-01-01T00:10:00
```

POST format

If *format* is `post`, the output will be also declared as *text/plain* and the structure will consist of: * a line with a URL where the request must be made, * a list of lines with the format declared in the FDSN Web Services specification to do a POST request.

If the request should be split in more than one datacenter, the blocks for every datacenter will be separated by a blank line and the structure will be repeated (URL and POST body).

```
http://ws.resif.fr/fdsnws/dataselect/1/query
4C KES28 * * 2010-01-01T00:00:00 2010-01-01T00:10:00
```

Alternative routes

Warning: As a rule of a thumb and in a normal case, the alternative addresses should only be used if there is no response from the authoritative data center.

If the *alternative* parameter is set, the service will return all the routes that match the requested criteria without paying attention to the priority. The client will be required to interpret the priority of the routes and to select the combination of routes that best fits their needs to request the information. The client needs also to take care of checking the information to detect overlapping routes, which will definitely occur when a primary and an alternative route are being reported for the same stream.

Note: It should be noted that the benefits of the "get" and "post" format outputs are almost nonexistent if alternative routes are included in the output, since the result should be parsed in order to operate on the different routes.

How to pass the parameters

In the case of performing a request via the `GET` method, the parameters must be given in the usual way. Namely,

```
http://server_url?key1=value1&key2=value2
```

But in the case that the parameters should be passed via a `POST` method, the following format is expected. The first lines can be used to pass the parameters not related to streams or timewindows (service, format, alternative) with one key=value clause per line. For instance,

```
service=station
```

For the six parameters used to select streams and timewindows, one stream/ timewindow pair is expected per line and the format must be:

```
net sta loc cha start end
```

If there is no defined time window, an empty string should be given as ‘’ or “”.

Warning: The separation of a request in more than one URL/parameters can be avoided by a client who performs an expansion of the wildcards before contacting this service. However, in some complex cases it could also happen that a stream is stored in two different data centers depending on the time window. In this case, it is unavoidable to split the request in more than one data center.

Abnormal responses

In addition to a `200 OK` status code for a successful request, other responses are possible, as shown in the *HTTP status codes returned by the Routing service*. These are essentially the same as for FDSN web services. Under error, maintenance or other unusual conditions a client may receive other HTTP codes generated by web service containers, and other intermediate web technology.

Table 3.2: HTTP status codes returned by the Routing service

Code	Description
200	OK, Successful request, results follow.
204	Request was properly formatted and submitted but no data matches the selection.
400	Bad request due to improper specification, unrecognized parameter, parameter value out of range, etc.
413	Request would result in too much data being returned or the request itself is too large. Returned error message should include the service limitations in the detailed description. Service limits should also be documented in the service WADL.
414	Request URI too large
500	Internal server error
503	Service temporarily unavailable, used in maintenance and error conditions

Information about the content of service

When the method *info* is invoked a description about the information handled by the Routing Service should be returned. The answer must be of MIME type *text/plain* and actually is a text-free output. However, in the first lines it is expected to be specified which information can we find by querying the service. For instance,

```
All Networks from XYZ institution
Stations in Indonesia
Stations in San Francisco

Other comments and descriptions that could be of interest of the user.
```

Any parameter passed to this method will be ignored.

CHAPTER

FOUR

DOCUMENTATION FOR DEVELOPERS

Routing module

Routing Service for EIDA.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

> **Copyright** 2014-2017 Javier Quinteros, GEOFON, GFZ Potsdam <geofon@gfz-potsdam.de>
>
> **License** GPLv3
>
> **Platform** Linux

Module author: Javier Quinteros <javier@gfz-potsdam.de>, GEOFON, GFZ Potsdam

`routing.`**`application`**(*environ*, *start_response*)
  Main WSGI handler. Process requests and calls proper functions.

`routing.`**`getParam`**(*parameters*, *names*, *default*, *csv=False*)
  Read a parameter and return its value or a default value.

`routing.`**`makeQueryGET`**(*parameters*)
  Process a request made via a GET method.

`routing.`**`makeQueryPOST`**(*postText*)
  Process a request made via a POST method.

Utils module

Classes to be used by the Routing WS for EIDA.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

> **Copyright** 2014-2017 Javier Quinteros, GEOFON, GFZ Potsdam <geofon@gfz-potsdam.de>
>
> **License** GPLv3
>
> **Platform** Linux

Module author: Javier Quinteros <javier@gfz-potsdam.de>, GEOFON, GFZ Potsdam

RoutingCache class

class `routeutils.utils.`**`RoutingCache`**(*routingFile=None*, *masterFile=None*, *config='routing.cfg'*)
  Manage routing information of streams read from an Arclink-XML file.

Platform Linux (maybe also Windows)

configArclink()

Connect via telnet to an Arclink server to get routing information.

Address and port of the server are read from the configuration file. The data is saved in the file `routing.xml`. Generally used to start operating with an EIDA default configuration.

Deprecated since version 1.1.

This method should not be used and the configuration should be independent from Arclink. Namely, the `routing.xml` file must exist in advance.

getRoute(*stream*, *tw*, *service='dataselect'*, *geoLoc=None*, *alternative=False*)

Return routes to request data for the stream and timewindow provided.

Based on a stream(s) and a timewindow returns all the needed information (URLs and parameters) to do the requests to different datacenters (if needed) and be able to merge the returned data avoiding duplication.

Parameters

- **stream** (*Stream*) – *Stream* definition including wildcards
- **tw** (*TW*) – Timewindow
- **service** (*str*) – Service from which you want to get information
- **geoLoc** (*geoRectangle*) – Rectangle to filter stations
- **alternative** (*bool*) – Specifies whether alternative routes should be included

Returns URLs and parameters to request the data

Return type *RequestMerge*

Raises RoutingException

getRouteDS(*service*, *stream*, *tw*, *geoLocation=None*, *alternative=False*)

Return routes to request data for the parameters specified.

Based on a *Stream* and a timewindow (*TW*) returns all the needed information (URLs and parameters) to request waveforms from different datacenters (if needed) and be able to merge it avoiding duplication.

Parameters

- **service** (*string*) – Specifies the service is being looked for
- **stream** (*Stream*) – *Stream* definition including wildcards
- **tw** (*TW*) – Timewindow
- **geoLocation** (*geoRectangle*) – Rectangle restricting the location of the station
- **alternative** (*bool*) – Specifies whether alternative routes should be included

Returns URLs and parameters to request the data

Return type *RequestMerge*

Raises RoutingException, ValueError

getRouteMaster(*n*, *tw*, *service='dataselect'*, *alternative=False*)

Look for a high priority *Route* for a particular network.

This would provide the flexibility to incorporate new networks and override the normal configuration.

Parameters

- **n** (*string*) – Network code
- **tw** (*TW*) – Timewindow

- **service** (*string*) – Service (e.g. dataselect)
- **alternative** (*Bool*) – Specifies whether alternative routes should be included

Returns URLs and parameters to request the data

Return type *RequestMerge*

Raises RoutingException

localConfig()
Return the local routing configuration.

Returns Local routing information in Arclink-XML format

Return type str

toXML(*foutput*, *nameSpace='ns0'*)
Export the RoutingCache to an XML representation.

update()
Read the routing data from the file saved by the off-line process.

All the routing information is read into a dictionary. Only the necessary attributes are stored. This relies on the idea that some other agent should update the routing data at a regular period of time.

updateAll()
Read the two sources of routing information.

updateMT()
Read the routes with highest priority and store them in memory.

All the routing information is read into a dictionary. Only the necessary attributes are stored. This relies on the idea that some other agent should update the routing file at a regular period of time.

updateVN()
Read the virtual networks defined.

Stations listed in each virtual network are read into a dictionary. Only the necessary attributes are stored. This relies on the idea that some other agent should update the routing file at a regular period of time.

vn2real(*stream*, *tw*)
Transform from a virtual network code to a list of streams.

Parameters

- **stream** (*Stream*) – requested stream including virtual network code.
- **tw** (*TW*) – time window requested.

Returns Streams and time windows of real network-station codes.

Return type list

Route class

class routeutils.utils.**Route**
Namedtuple defining a *Route*.

The attributes are service: service name address: a URL tw: timewindow priority: priority of the route

Platform Any

overlap(*otherRoute*)
Check if there is an overlap between this route and otherRoute.

Parameters **other** (*Stream*) – *Stream* which should be checked for overlaps

Returns Value specifying whether there is an overlap between this stream and the one passed as a parameter

Return type Bool

`toXML`(*nameSpace='ns0'*, *level=2*)
Export the Route to an XML representation.

Stream class

class `routeutils.utils.`**`Stream`**
Namedtuple representing a Stream.

It includes methods to calculate matching and overlapping of streams including (or not) wildcards. Components are the usual to determine a stream:

n: network s: station l: location c: channel

Platform Any

`overlap`(*other*)
Check if there is an overlap between this stream and other one.

Parameters **`other`** (*`Stream`*) – *`Stream`* which should be checked for overlaps

Returns Value specifying whether there is an overlap between this stream and the one passed as a parameter

Return type Bool

`strictMatch`(*other*)
Return a *reduction* of this stream to match what's been received.

Parameters **`other`** (*`Stream`*) – *`Stream`* which should be checked for overlaps

Returns *reduced* version of this *`Stream`* to match the one passed in the parameter

Return type *`Stream`*

Raises Exception

`toXMLclose`(*nameSpace='ns0'*, *level=1*)
Close the XML representation of a route given by toXMLopen.

`toXMLopen`(*nameSpace='ns0'*, *level=1*)
Export the stream to XML representing a route.

XML representation is incomplete and needs to be closed by the method toXMLclose.

Station class

class `routeutils.utils.`**`Station`**
Namedtuple representing a Station.

This is the minimum information which needs to be cached from a station in order to be able to apply a proper filter to the inventory when queries f.i. do not include the network name.

name: station name latitude: latitude longitude: longitude

Platform Any

geoRectangle class

class `routeutils.utils.`**`geoRectangle`**

Namedtuple representing a geographical rectangle.

minlat: minimum latitude maxlat: maximum latitude minlon: minimum longitude maxlon: maximum longitude

Platform Any

`contains`(*lat*, *lon*)

Check if the point belongs to the rectangle.

TW (timewindow) class

class `routeutils.utils.`**`TW`**

Namedtuple with methods to perform calculations on timewindows.

Attributes are: start: Start datetime end: End datetime

Platform Any

`difference`(*otherTW*)

Substract otherTW from this TW.

The result is a list of TW. This operation does not modify the data in the current timewindow.

Parameters **`otherTW`** (*`TW`*) – timewindow which should be substracted from this one

Returns Difference between this timewindow and the one in the parameter

Return type list of *`TW`*

`intersection`(*otherTW*)

Calculate the intersection between otherTW and this TW.

This operation does not modify the data in the current timewindow.

Parameters **`otherTW`** (*`TW`*) – timewindow which should be intersected with this one

Returns Intersection between this timewindow and the one in the parameter

Return type *`TW`*

`overlap`(*otherTW*)

Check if the otherTW is contained in this *`TW`*.

Parameters **`otherTW`** (*`TW`*) – timewindow which should be checked for overlapping

Returns Value specifying whether there is an overlap between this timewindow and the one in the parameter

Return type Bool

Examples

```
>>> y2011 = datetime(2011, 1, 1)
>>> y2012 = datetime(2012, 1, 1)
>>> y2013 = datetime(2013, 1, 1)
>>> y2014 = datetime(2014, 1, 1)
>>> TW(y2011, y2014).overlap(TW(y2012, y2013))
True
>>> TW(y2012, y2014).overlap(TW(y2011, y2013))
True
```

```
>>> TW(y2012, y2013).overlap(TW(y2011, y2014))
True
>>> TW(y2011, y2012).overlap(TW(y2013, y2014))
False
```

RouteMT class

RequestMerge class

class `routeutils.utils.`**`RequestMerge`**

Extend a list to group data from many requests by datacenter.

Platform Any

`append`(*service*, *url*, *priority*, *stream*, *tw*)

Append a new `Route` without repeating the datacenter.

Overrides the *append* method of the inherited list. If another route for the datacenter was already added, the remaining attributes are appended in *params* for the datacenter. If this is the first `Route` for the datacenter, everything is added.

Parameters

- **`service`** (*`str`*) – Service name (f.i., 'dataselect')
- **`url`** (*`str`*) – URL for the service (f.i., 'http://server/path/query')
- **`priority`** (*`int`*) – Priority of the Route (1: highest priority)
- **`stream`** (*`Stream`*) – Stream(s) associated with the Route
- **`start`** (*`datetime or None`*) – Start date for the Route
- **`end`** (*`datetime or None`*) – End date for the Route

`extend`(*listReqM*)

Append all the items in `RequestMerge` grouped by datacenter.

Overrides the *extend* method of the inherited list. If another route for the datacenter was already added, the remaining attributes are appended in *params* for the datacenter. If this is the first `Route` for the datacenter, everything is added.

Parameters **`listReqM`** (list of `RequestMerge`) – Requests from (posibly) different datacenters to be added

`index`(*service*, *url*)

Check for the service and url specified in the parameters.

This overrides the *index* method of the inherited list.

Parameters

- **`service`** (*`str`*) – Requests from (possibly) different datacenters to be added
- **`url`** (*`str`*) – Address of the service provided by a datacenter

Returns position in the list where the service and url specified can be found

Return type int

Raises ValueError

Wsgicomm module

Functions and resources to communicate via a WSGI module

Platform Linux

Module author: Javier Quinteros <javier@gfz-potsdam.de>, GEOFON, GFZ Potsdam

class `routeutils.wsgicomm.`**`Logs`**(*level=2*, *outstr=<open file '<stdout>'*, *mode 'w'>*)
Given a log level and a stream, redirect the output to the proper place.

Platform Linux

`setLevel`(*level*)
Set the level of the log.

Parameters **`level`** (`int`) – Log level (1: Error, 2: Warning, 3: Info, 4: Debug)

exception `routeutils.wsgicomm.`**`PlsRedirect`**(*url*)
Exception to signal that the web client must be redirected to a URL.

The constructor of the class receives a string, which is the URL where the web browser is going to be redirected.

exception `routeutils.wsgicomm.`**`WIClientError`**(**args*, ***kwargs*)
Exception to signal that an invalid request was received (400).

Platform Linux

exception `routeutils.wsgicomm.`**`WIContentError`**(**args*, ***kwargs*)
Exception to signal that no content was found (204).

Platform Linux

exception `routeutils.wsgicomm.`**`WIError`**(*status*, *body*, *verbosity=1*)
Exception to signal that an error occurred.

Platform Linux

exception `routeutils.wsgicomm.`**`WIInternalError`**(**args*, ***kwargs*)
Exception to signal that an internal server error occurred (500).

Platform Linux

exception `routeutils.wsgicomm.`**`WIServiceError`**(**args*, ***kwargs*)
Exception to signal that the service is unavailable (503).

Platform Linux

exception `routeutils.wsgicomm.`**`WIURIError`**(**args*, ***kwargs*)
Exception to signal that the URI is beyond the allowed limit (414).

Platform Linux

`routeutils.wsgicomm.`**`redirect_page`**(*url*, *start_response*)
Tell the web client through the WSGI module to redirect to a URL.

Platform Linux

`routeutils.wsgicomm.`**`send_dynamicfile_response`**(*status*, *body*, *start_response*)
Send a file (or file-like) object.

Caller must set the filename, size and content_type attributes of body.

Platform Linux

`routeutils.wsgicomm.`**`send_error_response`**(*status*, *body*, *start_response*)
Send a plain response in WSGI style.

Platform Linux

`routeutils.wsgicomm.`**`send_file_response`**(*status*, *body*, *start_response*)

Send a file (or file-like) object.

Caller must set the filename, size and content_type attributes of body.

Platform Linux

`routeutils.wsgicomm.`**`send_html_response`**(*status*, *body*, *start_response*)

Send an HTML response in WSGI style.

Platform Linux

`routeutils.wsgicomm.`**`send_nobody_response`**(*status*, *start_response*)

Send a plain response without body in WSGI style.

Platform Linux

`routeutils.wsgicomm.`**`send_plain_response`**(*status*, *body*, *start_response*)

Send a plain response in WSGI style.

Platform Linux

`routeutils.wsgicomm.`**`send_xml_response`**(*status*, *body*, *start_response*)

Send an XML response in WSGI style.

Platform Linux

PYTHON MODULE INDEX

r

routeutils.utils, 17
routeutils.wsgicomm, 22
routing, 17

A

append() (routeutils.utils.RequestMerge method), 22
application() (in module routing), 17

C

configArclink() (routeutils.utils.RoutingCache method), 18
contains() (routeutils.utils.geoRectangle method), 21

D

difference() (routeutils.utils.TW method), 21

E

extend() (routeutils.utils.RequestMerge method), 22

G

geoRectangle (class in routeutils.utils), 21
getParam() (in module routing), 17
getRoute() (routeutils.utils.RoutingCache method), 18
getRouteDS() (routeutils.utils.RoutingCache method), 18
getRouteMaster() (routeutils.utils.RoutingCache method), 18

I

index() (routeutils.utils.RequestMerge method), 22
intersection() (routeutils.utils.TW method), 21

L

localConfig() (routeutils.utils.RoutingCache method), 19
Logs (class in routeutils.wsgicomm), 23

M

makeQueryGET() (in module routing), 17
makeQueryPOST() (in module routing), 17

O

overlap() (routeutils.utils.Route method), 19
overlap() (routeutils.utils.Stream method), 20
overlap() (routeutils.utils.TW method), 21

P

PlsRedirect, 23

R

redirect_page() (in module routeutils.wsgicomm), 23
RequestMerge (class in routeutils.utils), 22
Route (class in routeutils.utils), 19
routeutils.utils (module), 17
routeutils.wsgicomm (module), 22
routing (module), 17
RoutingCache (class in routeutils.utils), 17

S

send_dynamicfile_response() (in module routeutils.wsgicomm), 23
send_error_response() (in module routeutils.wsgicomm), 23
send_file_response() (in module routeutils.wsgicomm), 24
send_html_response() (in module routeutils.wsgicomm), 24
send_nobody_response() (in module routeutils.wsgicomm), 24
send_plain_response() (in module routeutils.wsgicomm), 24
send_xml_response() (in module routeutils.wsgicomm), 24
setLevel() (routeutils.wsgicomm.Logs method), 23
Station (class in routeutils.utils), 20
Stream (class in routeutils.utils), 20
strictMatch() (routeutils.utils.Stream method), 20

T

toXML() (routeutils.utils.Route method), 20
toXML() (routeutils.utils.RoutingCache method), 19
toXMLclose() (routeutils.utils.Stream method), 20
toXMLopen() (routeutils.utils.Stream method), 20
TW (class in routeutils.utils), 21

U

update() (routeutils.utils.RoutingCache method), 19
updateAll() (routeutils.utils.RoutingCache method), 19
updateMT() (routeutils.utils.RoutingCache method), 19
updateVN() (routeutils.utils.RoutingCache method), 19

V

vn2real() (routeutils.utils.RoutingCache method), 19

W

WIClientError, 23

WIContentError, 23
WIError, 23
WIInternalError, 23
WIServiceError, 23
WIURIError, 23