Enhydris Documentation

Release b

National Technical University of Athens

May 28, 2015

Contents

Enhydris is a free database system for the storage and management of hydrological and meteorological data. It allows the storage and retrieval of raw data, processed time series, model parameters, curves and meta-information such as measurement stations overseers, instruments, events etc.

General documentation:

CHAPTER 1

About Enhydris

1.1 General

Enhydris is a system for the storage and management of hydrological and meteorological time series.

The database is accessible through a web interface, which includes several data representation features such as tables, graphs and mapping capabilities. Data access is configurable to allow or to restrict user groups and/or privileged users to contribute or to download data. With these capabilities, Enhydris can be used either as a public repository of free data or as a private system for data storage. Time series can be downloaded in plain text format that can be directly loaded to Hydrognomon, a free tool for analysis and processing of meteorological time series.

Enhydris is free software, available under the GNU Affero General Public License, and can run on UNIX (such as GNU/Linux) or Windows. Written in Python/Django, it can be installed on every operating system on which Python runs, including GNU/Linux and Windows. It is free software, available under the GNU General Public License version 3 or any later version. It is being used by openmeteo.org, Hydrological Observatory of Athens, Hydroscope, the Athens Water Supply Company, and WQ DREAMS.

Enhydris has several advanced features:

- It stores time series in a clever compressed text format in the database, resulting in using small space and high speed retrieval. However, the first and last few records of each time series are stored uncompressed, which means that the start and end date can be retrieved immediately, and appending a few records at the end can also be done instantly.
- It can work in a distributed way. Many organisations can install one instance each, but an additional instance, common to all organisations, can be setup as a common portal. This additional instance can be configured to replicate data from the databases of the organisations, but without the space-consuming time series, which it retrieves from the other databases on demand. A user can transparently use this portal to access the data of all participating organisations collectively.
- It offers access to the data through a webservice API. This is the foundation on which the above distributing features are based, but it can also be used so that other systems access the data.
- It has a security system that allows it to be used either in an organisational setting or in a public setting. In an organisational setting, there are priviliged users who have write access to all the data. In a public setting, users can subscribe, create stations, and add data for them, but they are not allowed to touch stations of other users.
- It is extensible. It is possible to create new Django applications which define geographical entity types besides stations, and reuse existing Enhydris functionality.

1.2 Presentations, documents, papers

Enhydris, Filotis & openmeteo.org: Free software for environmental management, by A. Christofides, S. Kozanis, G. Karavokiros, and A. Koukouvinos; FLOSS Conference 2011, Athens, 21 May 2011.

Enhydris: A free database system for the storage and management of hydrological and meteorological data, by A. Christofides, S. Kozanis, G. Karavokiros, Y. Markonis, and A. Efstratiadis; European Geosciences Union General Assembly 2011, Geophysical Research Abstracts, Vol. 13, Vienna, 8760, 2011.

CHAPTER 2

Installation and configuration

2.1 Download Enhydris

Download Enhydris from https://github.com/openmeteo/enhydris/ (if you are uncomfortable with git and github, click on the "Download ZIP" button).

2.2 Prerequisites

Prerequisite	Version
Python	2.6 [1]
PostgreSQL	[2]
PostGIS	1.4 [3]
GDAL	1.9
psycopg2	2.2 [4]
setuptools	0.6 [5]
pip	1.1 [5]
PIL with freetype	1.1.7 [6]
Dickinson	0.1.0 [7]
The Python modules listed in `requirements.txt`	See file

Note for production installations

These prerequisites are for development installations. For production installations you also need a web server.

[1] Enhydris runs on Python 2.6 and 2.7. It should also run on any later 2.x version. Enhydris does not run on Python 3.

[2] Enhydris should run on all supported PostgreSQL versions. In order to avoid possible incompatibilities with psycopg2, it is better to use the version prepackaged by your operating system when running on GNU/Linux, and to use the latest PostgreSQL version when running on Windows. If there is a problem with your version of PostgreSQL, email us and we'll check if it is easy to fix.

[3] Except for PostGIS, more libraries, namely geos and proj, are needed; however, you probably not need to worry about that, because in most GNU/Linux distributions PostGIS has a dependency on them and therefore they will be installed automatically, whereas in Windows the installation file of PostGIS includes them. Enhydris is known to run on PostGIS 1.4 and 1.5. It probably can run on later versions as well. It is not known whether it can run on earlier versions.

[4] psycopg2 is listed in `requirements.txt` together with the other Python modules. However, in contrast to them, it can be tricky to install (because it needs compilation and has a dependency on PostgreSQL client libraries),

and it is therefore usually better to not leave its installation to **pip**. It's better to install a prepackaged version for your operating system.

[5] setuptools and pip are needed in order to install the rest of the Python modules; Enhydris does not actually need it.

[6] PIL is not directly required by Enhydris, but by other python modules required my Enhydris. In theory, installing the requirements listed in `requirements.txt` will indirectly result in **pip** installing it. However, it can be tricky to install, and it may be better to not leave its installation to **pip**; it's better to install a prepackaged version for your operating system. It must be compiled with libfreetype support. This is common in Linux distributions. In Windows, however, the official packages are not thus compiled. One solution is to get the unofficial version from http://www.lfd.uci.edu/~gohlke/pythonlibs/. If there is any difficulty, Pillow might work instead of PIL.

[7] Dickinson is not required directly by Enhydris, but by pthelma, which is required by Enhydris and is listed in `requirements.txt`.

Example: Installing prerequisites on Debian/Ubuntu

These instructions are for Debian wheezy. For Ubuntu they are similar, except that the postgis package version may be different:

```
aptitude install python postgresql postgis postgresql-9.1-postgis \
    python-psycopg2 python-setuptools git python-pip python-imaging \
    python-gdal

# Install Dickinson
cd /tmp
wget https://github.com/openmeteo/dickinson/archive/0.1.0.tar.gz
tar xzf 0.1.0.tar.gz
cd dickinson-0.1.0
./configure
make
sudo make install

pip install -r requirements.txt
```

It is a good idea to use a virtualenv before running the last command, but you are on your own with that, sorry.

Example: Installing prerequisites on Windows

Important

We don't support Enhydris very well on Windows. We do provide instructions, and we do fix bugs, but honestly we can't install it; we get an error message related to "geos" at some point. Some people have had success by installing Enhydris using OSGeo4W, but we haven't tried it. So, if you face installation problems, we won't be able to help (unless you provide funding).

Also note that we don't think Enhydris on Windows can easily run on 64-bit Python or 64-bit PostgreSQL; the 32-bit versions of everything should be installed. This is because some prerequisites are not available for Windows in 64-bit versions, or they may be difficult to install. Such dependencies are PostGIS and some Python packages.

That said, we provide instructions below on how it should (in theory) be installed. If you choose to use OSGeo4W, some things will be different - you are on your own anyway.

Download and install the latest Python 2.x version from http://python.org/ (use the Windows Installer package).

Add the Python installation directory (such as `C:\Python27`) and its `Scripts` subdirectory (such as `C:\Python27\Scripts`) to the system path (right-click on My Computer, Properties, Advanced, Environment variables, under "System variables" double-click on Path, and add the two new directory names at the end, using semicolon to delimit them).

Download and install an appropriate PostgreSQL version from http://postgresql.org/ (use a binary Windows installer). Important: at some time the installer will create an operating system user and ask you to define a password for that user; keep the password; you will need it later.

Go to Start, All programs, PostgreSQL, Application Stack Builder, select your PostgreSQL installation on the first screen, then, on the application selection screen, select Spatial Extensions, PostGIS. Allow it to install (you don't need to create a spatial database at this stage).

Download and install psycopg2 for Windows from http://www.stickpeople.com/projects/python/win-psycopg/.

Download and install setuptools from http://pypi.python.org/pypi/setuptools (you probably need to go to http://pypi.python.org/pypi/setuptools#files and pick the .exe file that corresponds to your Python version).

Download and install PIL from http://www.lfd.uci.edu/~gohlke/pythonlibs/.

Download the latest dickinson DLL from http://openmeteo.org/downloads/ and put it in `C:\Windows\System32\dickinson.dll`.

Finally, open a Command Prompt and give the following commands inside the downloaded and unpacked `enhydris` directory:

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

2.3 Creating a spatially enabled database

You need to create a database user and a spatially enabled database (we use `enhydris_user` and `enhydris_db` in the examples below). Enhydris will be connecting to the database as that user. The user should not be a super user, not be allowed to create databases, and not be allowed to create more users.

GNU example

First, you need to create a spatially enabled database template. For PostGIS 2.0 or later (for earlier version refer to the GeoDjango instructions):

```
sudo -u postgres -s
createdb template_postgis
psql -d template_postgis -c "CREATE EXTENSION postgis;"
psql -d template_postgis -c \
   "UPDATE pg_database SET datistemplate='true' \
   WHERE datname='template_postgis';"
exit
```

The create the database:

```
sudo -u postgres -s
createuser --pwprompt enhydris_user
createdb --template template_postgis --owner enhydris_user \
   enhydris_db
exit
```

You may also need to edit your `pg_hba.conf` file as needed (under `/var/lib/pgsql/data/` or `/etc/postgresql/8.x/main/`, depending on your system). The chapter on client authentication of the PostgreSQL manual explains this in detail. A simple setup is to authenticate with username and password, in which case you should add or modify the following lines in `pg_hba.conf`:

```
local   all         all                               md5
host    all         all         127.0.0.1/32          md5
host    all         all         ::1/128               md5
```

Restart the server to read the new `pg_hba.conf` configuration. For example, in Ubuntu:

```
service postgresql restart
```

Windows example

Assuming PostgreSQL is installed at the default location, run these at a command prompt:

```
cd C:\Program Files\PostgreSQL\9.0\bin
createdb template_postgis
psql -d template_postgis -c "CREATE EXTENSION postgis;"
psql -d template_postgis -c "UPDATE pg_database SET datistemplate='true'
   WHERE datname='template_postgis';"
createuser -U postgres --pwprompt enhydris_user
createdb --template template_postgis --owner enhydris_user enhydris_db
```

At some point, these commands will ask you for the password of the operating system user.

2.4 Configuring Enhydris

In the directory `enhydris/settings`, copy the file `example.py` to `local.py`. Open `local.py` in an editor and make the following changes:

- Set `ADMINS` to a list of admins (the administrators will get all enhydris exceptions by mail and also all user emails, as generated by the contact application).
- Under `DATABASES`, set `NAME` to the name of the database, and `USER` and `PASSWORD` according to the user created above.

2.5 Initializing the database

In order to initialize your database and create the necessary database tables for Enhydris to run, run the following commands inside the `enhydris` directory:

```
python manage.py syncdb --settings=enhydris.settings.local --noinput
python manage.py migrate --settings=enhydris.settings.local dbsync
python manage.py migrate --settings=enhydris.settings.local hcore
python manage.py createsuperuser --settings=enhydris.settings.local
```

The above commands will also ask you to create a Enhydris superuser.

Confused by users?

There are operating system users, database users, and Enhydris users. PostgreSQL runs as an operating system user, and so does the web server, and so does Django and therefore Enhydris. Now the application (i.e. Enhydris/Django) needs a database connection to work, and for this connection it connects to the database as a database user. For the end users, that is, for the actual people who use Enhydris, Enhydris/Django keeps a list of usernames and passwords in the database, which have nothing to do with operating system users or database users. The Enhydris superuser created by the `./manage.py createsuperuser` command is such an Enhydris user, and is intended to represent a human.

Advanced Django administrators can also use alternative authentication backends, such as LDAP, for storing the Enhydris users.

2.6 Running Enhydris

Inside the `openmeteo/enhydris` directory, run the following command:

```
python manage.py runserver --settings=enhydris.settings.local 8088
```

The above command will start the Django development server and set it to listen to port 8088. If you then start your browser and point it to `http://localhost:8088/`, you should see Enhydris in action. Note that this only listens to the localhost; if you want it to listen on all interfaces, use `0.0.0.0:8088` instead.

To use Enhydris in production, you need to setup a web server such as apache. This is described in detail in Deploying Django.

2.7 Post-install configuration

2.7.1 Domain name

After you run Enhydris, logon as a superuser, visit the admin panel, go to `Sites`, edit the default site, and enter your domain name there instead of `example.com`. Emails to users for registration confirmation will appear to be coming from that domain. Restart the webserver after changing the domain name.

2.8 Settings reference

These are the settings available to Enhydris, in addition to the Django settings.

ENHYDRIS_FILTER_DEFAULT_COUNTRY
: When a default country is specified, the station search is locked within that country and the station search filter allows only searches in the selected country. If left blank, the filter allows all countries to be included in the search.

ENHYDRIS_FILTER_POLITICAL_SUBDIVISION1_NAME

ENHYDRIS_FILTER_POLITICAL_SUBDIVISION2_NAME
: These are used only if `FILTER_DEFAULT_COUNTRY` is set. They are the names of the first and the second level of political subdivision in a certain country. For example, Greece is first divided in 'districts', then in 'prefecture', whereas the USA is first divided in 'states', then in 'counties'.

ENHYDRIS_USERS_CAN_ADD_CONTENT
: This must be configured before syncing the database. If set to `True`, it enables all logged in users to add content to the site (stations, instruments and timeseries). It enables the use of user space forms which are available to all registered users and also allows editing existing data. When set to `False` (the default), only privileged users are allowed to add/edit/remove data from the db.

ENHYDRIS_SITE_CONTENT_IS_FREE
: If this is set to `True`, all registered users have access to the timeseries and can download timeseries data. If set to `False` (the default), the users may be restricted.

ENHYDRIS_TSDATA_AVAILABLE_FOR_ANONYMOUS_USERS
Setting this option to `True` will enable all users to download timeseries data without having to login first. The default is `False`.

ENHYDRIS_STORE_TSDATA_LOCALLY
Deprecated.

By default, this is `True`. If set to `False`, the installation does not store the actual time series records. The purpose of this setting is to be used together with the *dbsync* application, in order to create a website that contains the collected data (except time series records) of several other Enhydris installations (see the `hcore_remotesyncdb` management command). However, all this is under reconsideration.

ENHYDRIS_REMOTE_INSTANCE_CREDENTIALS
If the instance is configured as a data aggregator and doesn't have the actual data locally stored, in order to fetch the data from another instance a user name and password must be provided which correspond to a superuser account in the remote instance. Many instances can be configured using this setting, each with its own user/pass combination following this scheme:

```
ENHYDRIS_REMOTE_INSTANCE_CREDENTIALS = {
  'kyy.hydroscope.gr': ('myusername','mypassword'),
  'itia.hydroscope.gr': ('anotheruser','anotherpass')
}
```

ENHYDRIS_USE_OPEN_LAYERS
Set this to `False` to disable the map.

ENHYDRIS_MIN_VIEWPORT_IN_DEGS
Set a value in degrees. When a geographical query has bounds with dimensions less than `MIN_VIEWPORT_IN_DEGS`, the map will have at least a dimension of `MIN_VIEWPORT_IN_DEGS`². Useful when showing a single entity, such as a hydrometeorological station. Default value is 0.04, corresponding to an area approximately 4×4 km.

ENHYDRIS_MAP_DEFAULT_VIEWPORT
A tuple containing the default viewport for the map in geographical coordinates, in cases of geographical queries that do not return anything. Format is (minlon, minlat, maxlon, maxlat) where lon and lat is in decimal degrees, positive for north/east, negative for west/south.

ENHYDRIS_TS_GRAPH_CACHE_DIR
The directory in which timeseries graphs are cached. It is automatically created if it does not exist. The default is subdirectory `enhydris-timeseries-graphs` of the system or user temporary directory.

ENHYDRIS_TS_GRAPH_BIG_STEP_DENOMINATOR
ENHYDRIS_TS_GRAPH_FINE_STEP_DENOMINATOR
Chart options for time series details page. The big step represents the max num of data points to be plotted, default is 200. The fine step are the max num of points between main data points to search for a maxima, default is 50.

ENHYDRIS_SITE_STATION_FILTER
This is a quick-and-dirty way to create a web site that only displays a subset of an Enhydris database. For example, the database of http://deucalionproject.gr/db/ is the same as that of http://openmeteo.org/db/; however, the former only shows stations relevant to the Deucalion project, because it has this setting:

```
ENHYDRIS_SITE_STATION_FILTER = {'owner__id__exact': '9'}
```

ENHYDRIS_DISPLAY_COPYRIGHT_INFO
If `True`, the station detail page shows copyright information for the station. By default, it is `False`. If all the stations in the database belong to one organization, you probably want to leave it to `False`. If the database is going to be openly accessed and contains data that belongs to many owners, you probably want to set it to `True`.

ENHYDRIS_WGS84_NAME

Sometimes Enhydris displays the reference system of the co-ordinates, which is always WGS84. In some installations, it is desirable to show something other than "WGS84", such as "ETRS89". This parameter specifies the name that will be displayed; the default is WGS84.

This is merely a cosmetic issue, which does not affect the actual reference system used, which is always WGS84. The purpose of this parameter is merely to enable installations in Europe to display "ETRS89" instead of "WGS84" whenever this is preferred. Given that the difference between WGS84 and ETRS89 is only a few centimeters, which is considerably less that the accuracy with which station co-ordinates are given, whether WGS84 or ETRS89 is displayed is actually irrelevant.

CHAPTER 3

Copyright and credits

Enhydris is

The database of Enhydris was originally written by Antonis Christofides of the National Technical University of Athens. The rest of Enhydris was originally developed by Andreas Loupasakis and Seraphim Mellos of Indifex.

Enhydris was funded by the Ministry of Environment of Greece as part of the Hydroscope project.

Reference:

CHAPTER 4

The database

4.1 Main principles

The Enhydris database is implemented in PostgreSQL. While the implementation of the database is through Django's object-relational mapper, which is more or less RDBMS-independent, Enhydris uses PostgreSQL's geographic features, so it is not portable. It also uses some custom PostgreSQL code for storing timeseries (however this is likely to change in the future).

In Django parlance, a *model* is a type of entity, which usually maps to a single database table. Therefore, in Django, we usually talk of models rather than of database tables, and we design models, which is close to a conceptual database design, leaving it to Django's object-relational mapper to translate to the physical. In this text, we also speak more of models than of tables. Since a model is a Python class, we describe it as a Python class rather than as a relational database table. If, however, you feel more comfortable with tables, you can generally read the text understanding that a model is a table.

If you are interested in the physical structure of the database, you need to know the model translation rules, which are quite simple:

- The name of the table is the lower case name of the model, with a prefix. The prefix for the core of the database is `hcore_`. (More on the prefix below).
- Tables normally have an implicit integer id field, which is the primary key of the table.
- Table fields have the same name as model attributes, except for foreign keys.
- Foreign keys have the name of the model attribute suffixed with `_id`.
- When using multi-table inheritance, the primary key of the child table is also a foreign key to the id field of the parent table. The name of the database column for the key of the child table is the lower cased parent model name suffixed with `_ptr_id`.

There are two drawings that accompany this text: the drawing for the `conceptual data model`, and the drawing for the `physical data model`. You should avoid looking at the physical data model; it is cluttered and confusing, since it is machine-generated. It is only provided for the benefit of those who are not comfortable with Django's object-relational mapping. However, it is best to learn to read the conceptual data model; if you become acquainted with the Django's object-relational mapping rules listed above, you will be able to write SQL commands effortlessly, by using these rules in your head. The drawing of the physical data model is also far more likely to contain errors or to be outdated than the drawing and documentation for the conceptual data model.

The core of the Enhydris database is a list of measuring stations, with additional information such as instruments, photos, videos, and so on, and the hydrological and meteorological time series stored for each measuring station. This can be used in or assisted by many more applications, which may or may not be needed in each setup. A billing system is needed for agencies that charge for their data, but not for those who offer them freely or only internally. Some organisations may need to develop additional software for managing aqueducts, and some may not. Therefore,

the core is kept as simple as possible. The core database tables use the `hcore_` prefix. Other applications use another prefix. The name of a table is the lowercased model name preceeded by the prefix. For example, the table that corresponds to the *`Gentity`* model is `hcore_gentity`.

Multilinguality

Originally, the database was designed in order to be multilingual, that is, so that the content could be stored in an unlimited number of languages. The django-multilingual framework was used for this purpose. However, django-multilingual bugs slowed development too much, and it was decided to go for a more modest solution: texts are simply stored in two languages: the local language and the alternative language. For example, for a description, there are the "descr" field and the "descr_alt" field. Which languages are "descr" and "descr_alt" depends on the installation. For example, we use Greek as the local language and English as the alternative language.

We hope to get rid of this, but this will involve fixing django-multilingual or using another multilingual framework.

When any field in the API is marked as being multilingual, it means that it is accompanied by an additional identical field that has "_alt" appended to its name. (It also means that, instead, it should be defined in a `Translation` class nested in the model class, as would be the case if django-multilingual were used.)

4.2 Lookup tables

Lookup tables are those that are used for enumerated values. For example, the list of variables is a lookup table. Most lookup tables in the Enhydris database have three fields: *id*, *descr*, and *short_descr*, and they all inherit the following abstract base class:

class `enhydris.hcore.models.`**Lookup**

This class contains the common attribute of the lookup tables:

descr

A *multilingual* character field with a descriptive name.

Most lookup tables are described in a relevant section of this document, where their description fits better; for example, *`StationType`* is described at Section *Station and its related models*.

4.3 Lentities

The Lentity is the superclass of people and groups. For example, a measuring station can belong either to an organisation or an individual. Lawyers use the word "entity" to refer to individuals and organisations together, but this would create confusion because of the more generic meaning of "entity" in computing; therefore, we use "lentity", which is something like a legal entity. The lentity hierarchy is implemented by using Django's multi-table inheritance.

class `enhydris.hcore.models.`**Lentity**

remarks

A *multilingual* text field of unlimited length.

class `enhydris.hcore.models.`**Person**

last_name

first_name

middle_names

initials
The above four are all *multilingual* character fields. The `initials` contain the initials without the last name. For example, for Antonis Michael Christofides, `initials` would contain the value "A. M.".

class enhydris.hcore.models.**Organization**

name

acronym
`name` and `acronym` are both *multilingual* character fields.

4.4 Gentity and its direct descendants: Gpoint, Gline, Garea

A Gentity is a geographical entity. Examples of gentities (short for geographical entities) are measuring stations, cities, boreholes and watersheds. A gentity can be a point (e.g. stations and boreholes), a surface (e.g. lakes and watersheds), a line (e.g. aqueducts), or a network (e.g. a river). The gentities implemented in the core are measuring stations and water basins. The gentity hierarchy is implemented by using Django's multi-table inheritance.

class enhydris.hcore.models.**Gentity**

name
A *multilingual* field with the name of the gentity, such as the name of a measuring station. Up to 200 characters.

short_name
A *multilingual* field with a short name of the gentity. Up to 50 characters.

remarks
A *multilingual* field with general remarks about the gentity. Unlimited length.

water_basin
The `water basin` where the gentity is.

water_division
The water division in which the gentity is. Foreign key to `WaterDivision`.

political_division
The country or other political division in which the gentity is. Foreign key to `PoliticalDivision`.

class enhydris.hcore.models.**Gpoint**(*Gentity*)

point
This is a GeoDjango PointField that stores the 2-d location of the point.

srid
Specifies the reference system in which the user originally entered the co-ordinates of the point. Valid *srid*'s are registered at http://www.epsg-registry.org/. See also http://itia.ntua.gr/antonis/technical/coordinate-systems/.

approximate
This boolean field has the value `True` if the horizontal co-ordinates are approximate. This normally means that the user who specified the co-ordinates did not really know the location of the point, but for convenience placed it somewhere visually so that the GIS system can have a rough idea of where to show it and e.g. in which basin it is.

altitude

asrid
These attributes store the altitude. *asrid* specifies the reference system, which defines how *altitude* is to be understood. *asrid* can be empty, in which case, *altitude* is given in metres above mean sea level.

class enhydris.hcore.models.**Gline**(*Gentity*)

gpoint1

gpoint2
The starting and ending points of the line; foreign keys to *Gpoint*.

length
The length of the line in meters.

class enhydris.hcore.models.**Garea**(*Gentity*)

area
The size of the area in square meters.

4.5 Additional information for generic gentities

This section describes models that provide additional information about gentities.

class enhydris.hcore.models.**PoliticalDivision**(*Garea*)
From an administrative point of view, the world is divided into countries. Each country is then divided into further divisions, which may be called states, districts, counties, provinces, prefectures, and so on, which may be further subdivided. Greece, for example, is divided in districts, which are subdivided in prefectures. How these divisions and subdivisions are named, and the way and depth of subdividing, differs from country to country.

PoliticalDivision is a recursive model that represents such political divisions. The top-level political division is a country, and lower levels differ from country to country.

parent
For top-level political divisions, that is, countries, this attribute is null; otherwise, it points to the containing political division.

code
For top-level political divisions, that is, countries, this is the two-character ISO 3166 country code. For lower level political divisions, it can be a country-specific division code; for example, for US states, it can be the two-character state code. Up to five characters.

class enhydris.hcore.models.**WaterDivision**(*Garea*)
A water division is a collection of basins. Water divisions may be used for administrative purposes, each water division being under the authority of one organisation or organisational division. Usually a water division consists of adjacent basins or of nearby islands or both.

class enhydris.hcore.models.**WaterBasin**(*Garea*)
A water basin.

parent
If this is a subbasin, this field points to the containing water basin.

water_division
The `water district` in which the water basin is.

class enhydris.hcore.models.**GentityAltCodeType**(*Lookup*)
The different kinds of codes that a gentity may have; see *GentityAltCode* for more information.

class enhydris.hcore.models.**GentityAltCode**
While each gentity is automatically given an id by the system, some stations may also have alternative codes. For example, in Greece, if a database contains a measuring station that is owned by a specific organisation, the station has the id given to it by the database, but in addition it may have a code assigned by the organisation; some also have a code created by older inter-organisational efforts to create a unique list of stations in Greece; and some also have a WMO code. This model therefore stores alternative codes.

gentity
A foreign key to *Gentity*.

type
The type of alternative code; one of those listed in *GentityAltCodeType*.

value
A character field with the actual code.

class enhydris.hcore.models.**FileType**(*Lookup*)
A lookup that contains one additional field:

mime_type
The mime type, like `image/jpeg`.

class enhydris.hcore.models.**GentityFile**
This model stores general files for the gentity. For examples, for measuring stations, it can be photos, videos, sensor manuals, etc.

descr
A *multilingual* short description or legend of the file.

remarks
Multilingual remarks of unlimited length.

date
For photos, it should be the date the photo was taken. For other kinds of files, it can be any kind of date.

file_type
The type of the file; a foreign key to *FileType*.

content
The actual content of the file; a Django FileField. Note that, for generality, images are also stored in this attribute, and therefore they don't use an ImageField, which means that the few facilities that ImageField offers are not available.

class enhydris.hcore.models.**EventType**(*Lookup*)
Stores types of events.

class enhydris.hcore.models.**GentityEvent**
An event is something that happens during the lifetime of a gentity and needs to be recorded. For example, for measuring stations, events such as malfunctions, maintenance sessions, and extreme weather phenomena observations can be recorded and provide a kind of log.

gentity
The *Gentity* to which the event refers.

date
The date of the event.

type
The *EventType*.

user
The username of the user who entered the event to the database.

report
A report about the event; a text field of unlimited length.

4.6 Station and its related models

class enhydris.hcore.models.**StationType**(*Lookup*)
The station type, such as "meteorological" or "stage measuring".

class enhydris.hcore.models.**Station**(*Gpoint*)

owner
The *Lentity* that owns the station.

type
The *StationType*.

is_active
A boolean field showing whether the station is operating.

is_automatic
A boolean field showing whether the station is automatic.

start_date

end_date
An optional pair of dates indicating when the station started and stopped working.

overseers
The overseers are the persons who are or have been responsible for each meteorological station in the past. In the case of traditional (not automatic) stations, this means the weather observers. At a given time, each station has only one observer. This is a many-to-many field, through model *Overseer*.

class enhydris.hcore.models.**Overseer**

station
A foreign key to *Station*.

person
A foreign key to *Person*.

is_current
A boolean value indicating whether this person is the current observer. For current overseers, the *end_date* below must be null; however, a null end_date could also mean that the end_date is unknown, not necessarily that the overseer is the current overseer.

start_date

end_date

class enhydris.hcore.models.**InstrumentType**(*Lookup*)
The instrument type, such as "Thermometer".

class enhydris.hcore.models.**Instrument**
A measuring instrument or sensor that belongs to a station.

station
The *Station* to which the instrument belongs.

type
The *InstrumentType*.

name
A *multilingual* field with a descriptive name.

remarks
A *multilingual* field with remarks of unlimited length.

manufacturer
The name of the manufacturer. For simplicity, this is not a foreign key to *Organization*; this would be overkill.

model
The model name.

is_active
A boolean indicating whether the instrument is operative.

start_date

end_date
The dates of start and end of operation.

4.7 Time series and related models

class enhydris.hcore.models.**Variable**(*Lookup*)
This model stores a variable, such as "precipitation", "evaporation", "temperature" etc.

class enhydris.hcore.models.**UnitOfMeasurement**(*Lookup*)
This model stores a unit of measurement. In addition to *Lookup* fields, it has the following additional fields:

symbol
The symbol used for the unit, in UTF-8 plain text.

variables
A many-to-many relationship to *Variable*.

class enhydris.hcore.models.**TimeZone**
This model stores time zones.

code
The code name of the time zone, such as CET or UTC.

utc_offset
A number, in minutes, with the offset of the time zone from UTC. For example, CET has a utc_offset of 60, whereas CDT is -300. This model only stores time zones with a constant utc offset, and not time zones with variable offsets. For example, we don't store CT (North American Central Time), because this is different in summer and in winter; instead, we store CST (Central Standard Time) and CDT (Central Daylight Time), which are the two occurrences of CT. The time stamps of a given time series may not observe summer time; they must always have the same utc offset throught the time series.

class enhydris.hcore.models.**TimeStep**(*Lookup*)
This model holds time steps. The *descr* attribute inherited by *Lookup* holds a descriptive name for the time step, such as "daily" or "monthly". The model has two additional attributes:

length_minutes

length_months
One of these two attributes must be zero. For example, a daily time step has length_minutes=1440 and length_months=0; an annual time step has length_minutes=0 and length_months=12.

class enhydris.hcore.models.**Timeseries**
This model holds information, but not the actual data, of a time series.

gentity
The *Gentity* to which the time series refers.

variable
The *Variable* of the time series.

unit_of_measurement
The *UnitOfMeasurement*.

name
A descriptive name for the time series.

precision
An integer specifying the precision of the values of the time series, in number of decimal digits. It can be negative; for example, a precision of -2 indicates that the values are accurate to the hundred, ex. 100, 200 etc.

time_zone
The *TimeZone* in which the time series' timestamps are.

remarks
A text field of unlimited length.

instrument
The instrument that measured the time series; a foreign key to *Instrument*. This can be null, as there are time series that are not measured by instruments, as are, for example, time series resulting from processing of other time series.

hidden
A boolean field to control the visibility of timeseries in related pages.

The rest of the attributes of the *Timeseries* model describe the time step and they are several:

time_step
nominal_offset_minutes
nominal_offset_months
actual_offset_minutes
actual_offset_months
The *time_step* is a foreign key to *TimeStep*. Some time series are completely irregular; in that case, *time_step* (and all other time step related attributes) is null. Otherwise, it contains an appropriate time step. For an explanation of the other four attributes, see the timeseries.TimeStep class. *actual_offset_minutes* and *actual_offset_months* must always be present if the time step is not null. The nominal offset attributes may, however, be null, if the time series is not strict, that is, if it does have a time step, but that time step contains irregularities. As an example, a time series measured by an automatic meteorological station every ten minutes will usually have a nominal offset of 0 minutes, which means the timestamps will end in :10, :20, :30, etc; but a clock error or a setup error could result in the timestamps ending in :11, :21, :31 for a brief period of time. In that case, we say that the time series has a nonstrict time step of 10 minutes, which means it has no specific nominal offset.

The time series records are stored in the ts_records table, the format of which is documented in pthelma. Although this table corresponds to a Django model, the existence of that model (which is a bit hacked and can run only on PostgreSQL) is only a means to create the table. The Django model should never be used to access the table; instead, the pthelma.timeseries.Timeseries methods read_from_db(), write_to_db(), and

`append_to_db()`, should be used. (It is also likely that these internals will change in the future, and the time series records will be stored by a Django FileField in the *`Timeseries`* table.

CHAPTER 5

Webservice API

5.1 Overview

Normally the web pages of Enhydris are good if you are a human; but if you are a computer (a script that creates stations, for example), then you need a different interface. For that purpose, Enydris offers an API through HTTP, through which applications can communicate. For example, http://openmeteo.org/stations/d/1334/ shows you a weather station in human-readable format; http://openmeteo.org/api/Station/1334/ provides you data on the same station in machine-readable format.

Important

The Webservice API might change heavily in the future. If you make any use of the API, it is very important that you stay in touch with us so that we take into account your backwards compatibility needs. Otherwise your applications might stop working one day.

The Webservice API is a work in progress: it was originally designed in order to provide the ability to replicate the data from one instance to another over the network. It was later extended to provide the possibility to create timeseries through a script. New functions are added to it as needed.

5.2 Client authentication

Some of the API functions are provided freely, while others require authentication. An example of the latter are functions which alter data; another example is data which are protected and need, for example, a subscription in order to be accessed. In such cases of restricted access, HTTP Basic authentication is performed.

Note: Using HTTP Basic Authentication with apache and mod_wsgi requires you to add the `WSGIPassAuthorization On` directive to the server or vhost config, otherwise the application cannot read the authentication data from HTTP_AUTHORIZATION in request.META. See: WSGI+BASIC_AUTH.

5.3 Generic API calls

API calls are accessible under the `/api/` url after which you just fill in the model name of the model you want to request. For example, to request all the stations you must provide the url `http://base-address/api/Station/`; the format in which the data will be returned depends on the HTTP `Accept` header. The same goes for the rest of the enhydris models (e.g. `/api/Garea/`, `/api/Gentity/`

etc). There is also the ability to request only one object of a specific type by appending its `id` in the url like this: `http://base-address/api/Station/1000/`.

See the *data model reference* for information on the models.

5.4 Creating new time series and stations

To create a new time series, you POST `/api/Timeseries/`; you must pass an appropriate csrf_token and a session id (you must be logged on as a user who has permission to do this), and pass the data in an appropriate format, such as JSON. Likewise, you can create new stations by POSTing `/api/Station/`; you can also delete stations and time series, and you can edit stations.

If you program in Python, you should use Pthelma's enhydris_api module. Otherwise, you should read its code to see more concrete examples of how to use the API.

5.5 Appending data to a time series

To append data to a time series, you PUT `api/tsdata`. See the code of loggertodb for an example of how to do this.

5.6 Timeseries data and GentityFile

At `http://base-address/api/tsdata/id/` (where `id` is the actual id of the timeseries object) you can get the timeseries data in text format.

Backwards-compatibility

The API implementation was changed in several changesets, starting with 639e4c810457. Before that, django-piston was being used for the api; it was changed to django-rest-framework.

Not all API features have been reimplemented. Notably, piston's output could be used with Django's loaddata management command to load data to an empty instance; this is no longer possible, because the returned objects do not contain a "model" attribute.

Furthermore, there was also the possibility to get gentity files at *http://base-address/api/gfdata/id*' (where `id` was the actual id of the GentityFile object). Finally, there was the "station information and lists" feature, documented below:

(Temporarily?) obsolete documentation on station information and lists

There are also some more calls which provide station details in a more human readable format, including a station's geodata which may be used by 3rd party application to incorporate the displaying of enhydris stations in their maps. These API calls reside under the `/api/Station/info/` url and are similar to the ones above. If you do not specify any additional parameters, you get information for all Stations hosted in Enhydris and if you want the details for a specific station, you just need to append its id to the end of the url like above (eg `/api/Station/info/1000`). See `models.Gentity` and `models.Station` for a description of the meaning of the fields.

There is also another feature which enables users to request a sublist of stations by providing the station ids in a comma separated list by using the `/api/Station/info/list` url. This call supports only the POST method and the comma separated list must be given under the varible name `station_list`. For example:

```
curl -X POST -d "station_list=10001,10002,10003" http://openmeteo.org/db/api/Station/info/list/
```

5.7 Cached time series data

At `http://base-address/timeseries/data/?object_id=id` (where `id` is the actual id of the time series object) you can get some time series data from specific positions (timestamps) as well as statistics and chart data. Data is cached so no need to read the entire time series and usually information is delivered fast.

Cached time series data are being used to display time series previews in time series detail pages. Also there are used for charting like in:

> http://openmeteo.org/db/chart/ntuastation/

The response is a JSON object. An example is the following:

```
{
  "stats": {"min_tstmp": 1353316200000,
            "max": 6.0,
            "max_tstmp": 979495200000,
            "avg": 0.0094982613015400109,
            "vavg": null,
            "count": 10065,
            "last_tstmp": 1353316200000,
            "last": 0.0,
            "min": 0.0,
            "sum": 95.600000000000207,
            "vectors": [0, 0, 0, 0, 0, 0, 0, 0],
            "vsum": [0.0, 0.0]},
  "data": [[911218200000, "0.0", 1],
           [913349400000, "4.8", 3551],
           ...,
           [1350248400000, "0.0", 710001],
           [1353316200000, "0.0", 715149]]
}
```

"stats" An object holding statistics for the given interval (see bellow)

"last" Last value observed for the given interval

"last_tstmp" The timestamp for the last value

"max" Is the maximum value observed for the given interval (see bellow)

"max_tstmp" The timestamp where the maximum value is observed

"min" The minimum value for the given interval

"min_tstmp" The timestamp where minimum value is observed

"avg" The average value for the given interval

"vavg" A vector average in decimal degrees for vector variables such as wind direction etc.

"count" The actual number of records used for statistics

"sum" The sum of values for the given interval

"vsum" Two components of sum (vector sum) Sx, Sy, computed by the cosines, sinus.

"vectors" The percentage of vector variable for eight distinct directions (N, NE, E, SE, S, SW, W and NW).

"data" An object holding an array of charting values. Each item of the array holds [timestamp, value, index]. Timestamp is a javascript timestamp, value if a floating point number or null, index is the actual index of the value in the whole time series records.

You have to specify at least the object_id GET parameter in order to obtain some data. The default time interval is the whole time series. In the case of the whole time series a rough image of the time series is displayed which is not precise. Statistics also can be no precise.

In example for 10-minute time step time series, chart and statistics can be precise for intervals of one month the most.

Besides `object_id` some other parameters can be given as GET parameters to specify the desired interval etc:

start_pos an index number specifying the begining of an interval. Index can be zero (0) for the begining of the time series or at most last record number minus one.

end_pos an index number specifying the end of an interval.

last

A string defining an interval from a pre-defined set:

- day
- week
- month
- year
- moment (returns one value only for the last moment)
- hour
- twohour

By default the end of the interval is the end of the time series. If time-series is auto-updated it shows the last measurements.

date Can be used in conjuction with the *last* parameter to display in interval beginning at the specified date. Date format: yyyy-mm-dd

time Can be used in conjuction with *last* and *date* parameters to specify the beginning time of the interval. Accepted format: HH:MM

exact_datetime A boolean parameter (set to true to activate). Specifies that date times should be existing in time series record or else it returns null. If not activated, it returns the closest periods with data to the specified interval.

start_offset An offset in minutes for the beginning of the interval. It can be used i.e. to exclude the first value of a daily interval, so the statistics are computed correct i.e. from 144 10-min values rather than 145 values (e.g. from 00:10 to 24:00 rather than 00:00 to 24:00). Suggested value for a ten minute time series is 10

vector A boolean parameter. Set to 'true' to activate. Then vector statistics are being calculated.

jsoncallback=? If you're running into the Same Origin Policy, which doesn't (normally) allow ajax requests to cross origins you should add the GET parameter above to obtain the cached time series data set.

A full example to get some daily values for a time series:

https://openmeteo.org/db/timeseries/data/?object_id=230&last=day&exact_datetime=true&date=2012-11-01&time=00:00

Contributed applications:

CHAPTER 6

dbsync — Database Syncing

The dbsync module implements the database replication and synchronization features. The core part of this module is the syncdb management command which takes care of fetching and installing remote objects from JSON files using the *Webservice API*.

Note

The dbsync application is currently barely working and should be rewritten.

6.1 DBSync Objects

Each instance of the `Database` class represents a remote enhydris instance. Once such an object has been added to the local database, then the remote instance it refers to can be used in the replication routine.

class dbsync.**Database**(*name*, *ip_address*, *hostname*, *descr*)

name
This is the name of the database. It's not mandatory that it's the same to the actual name of the database. This is only used for local reference.

ip_address
This field should contain the ip of the host that holds the remote enhydris instance.

hostname
This field must contain the FQDN from which the enhydris instance is accessible (this is especially required when using vhosts on a server so that the replication script knows which vhost uses which database).

Note: A fully qualified domain name (FQDN), sometimes referred to as an absolute domain name, is a domain name that specifies its exact location in the tree hierarchy of the Domain Name System (DNS). It specifies all domain levels, including the top-level domain, relative to the root domain. A fully qualified domain name is distinguished by this absoluteness in the name space.

descr
This is a textfield that holds the description for the specific database.

6.2 DBSync Management Command

The core functionality of the DBSync module is to provide a management command with which one can replicate completely a remote instance (or multiple remote instances) of the enhydris web application. The replication script can also update existing entries with changes when run multiple consecutive times but doesn't handle item deletion.

The code for the replication scripts resides under the `enhydris/dbsync/management/commands/` directory, inside the `hcore_remotesyncdb.py` file. You can check out the available options for the script by issuing the following command:

```
# ./manage.py hcore_remotesyncdb -h

Usage: ./manage.py hcore_remotesyncdb [options]

This command is used to synchronize the local database using data from a
remote instance

Options:
  -v VERBOSITY, --verbosity=VERBOSITY
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=all 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           Print traceback on exception
  -r REMOTE, --remote=REMOTE
                        Remote instance to sync from
  -p PORT, --port=PORT  Specify custom port. Default is 80.
  -a APP, --app=APP     Application which should be synced
  -e EXCLUDE, --exclude=EXCLUDE
                        State which models of the apps you want excluded from
                        the sync
  -f, --fetch-only      Doesn't actually submit any changes, just fetches
                        remote dumps and saves them locally.
  -w CWD, --work-dir=CWD
                        Define the tmp dir in which all temporary files will
                        be stored
  -N, --no-backups      Default behaviour is to take a backup of the local db
                        before doing any changes. This overrides this
                        behavior.
  -s, --skip            If skip is specified, then syncing will skip any
                        problems continue execution. Default behavior is to
                        halt on all errors.
  -R, --resume          With resume, no files are fetched but the local ones
                        are used.
  -S, --silent          Suppress all log messages
  --version             show program's version number and exit
  -h, --help            show this help message and exit
```

The most important command line options are the `-a` and `-r` which are used to specify which application you want to replicate (in our case `hcore`) and which is the remote instance from which the data should be pulled. A sample execution of the replication script from the command line should look something like this:

```
# ./manage.py hcore_remotesyncdb -a hcore -r itia.hydroscope.gr -e UserProfile
/usr/local/lib/python2.6/dist-packages/django_registration-0.7-py2.6.egg/registration/models.py:4:
DeprecationWarning: the sha module is deprecated; use the hashlib module instead
Checking port availability on host 147.102.160.28, port 80
Remote host is up. Continuing with the sync.
The following models will be synced: ['EventType', 'FileType', 'Garea',
'Gentity', 'GentityAltCode', 'GentityAltCodeType', 'GentityEvent',
'GentityFile', 'Gline', 'Gpoint', 'Instrument', 'InstrumentType', 'Lentity',
'Organization', 'Overseer', 'Person', 'PoliticalDivision', 'Station',
'StationType', 'TimeStep', 'TimeZone', 'Timeseries', 'UnitOfMeasurement',
'Variable', 'WaterBasin', 'WaterDivision']
The following models will be excluded ['UserProfile']
Syncing model EventType
     - Downloading EventType fixtures : done
Syncing model FileType
     - Downloading FileType fixtures : done
Syncing model Garea
     - Downloading Garea fixtures : done
Syncing model Gentity
     - Downloading Gentity fixtures : done
Syncing model GentityAltCode
     - Downloading GentityAltCode fixtures : done
Syncing model GentityAltCodeType
     - Downloading GentityAltCodeType fixtures : done
Syncing model GentityEvent
     - Downloading GentityEvent fixtures : done
Syncing model GentityFile
     - Downloading GentityFile fixtures : done
Syncing model Gline
     - Downloading Gline fixtures : done
Syncing model Gpoint
     - Downloading Gpoint fixtures : done
Syncing model Instrument
     - Downloading Instrument fixtures : done
Syncing model InstrumentType
     - Downloading InstrumentType fixtures : done
Syncing model Lentity
     - Downloading Lentity fixtures : done
Syncing model Organization
     - Downloading Organization fixtures : done
Syncing model Overseer
     - Downloading Overseer fixtures : done
Syncing model Person
     - Downloading Person fixtures : done
Syncing model PoliticalDivision
     - Downloading PoliticalDivision fixtures : done
Syncing model Station
     - Downloading Station fixtures : done
Syncing model StationType
     - Downloading StationType fixtures : done
Syncing model TimeStep
     - Downloading TimeStep fixtures : done
Syncing model TimeZone
     - Downloading TimeZone fixtures : done
Syncing model Timeseries
     - Downloading Timeseries fixtures : done
Syncing model UnitOfMeasurement
     - Downloading UnitOfMeasurement fixtures : done
```

```
Syncing model Variable
     - Downloading Variable fixtures : done
Syncing model WaterBasin
     - Downloading WaterBasin fixtures : done
Syncing model WaterDivision
     - Downloading WaterDivision fixtures : done
Creating Generic objects
Finished with Generic objects
Installing fixtures from file EventType.json
Installing fixtures from file FileType.json
Installing fixtures from file Gentity.json
Installing fixtures from file Garea.json
Installing fixtures from file GentityAltCode.json
Installing fixtures from file GentityAltCodeType.json
Installing fixtures from file GentityEvent.json
Installing fixtures from file GentityFile.json
Installing fixtures from file Gline.json
Installing fixtures from file Gpoint.json
Installing fixtures from file Instrument.json
Installing fixtures from file InstrumentType.json
Installing fixtures from file Lentity.json
Installing fixtures from file Organization.json
Installing fixtures from file Overseer.json
Installing fixtures from file Person.json
Installing fixtures from file PoliticalDivision.json
Installing fixtures from file Station.json
Installing fixtures from file StationType.json
Installing fixtures from file TimeStep.json
Installing fixtures from file TimeZone.json
Installing fixtures from file Timeseries.json
Installing fixtures from file UnitOfMeasurement.json
Installing fixtures from file Variable.json
Installing fixtures from file WaterBasin.json
Installing fixtures from file WaterDivision.json
Reinitializing foreign keys: done
Successfully installed 7319 objects from 26 fixtures.
```

The command above, replicates all remote data except for the UserProfiles (defined using the `-e|--exclude` option) keeping all data and foreign keys intact but without preserving the object ids. If run multiple times, the script can also update existing entries along with adding new ones. It's important to note that when replicating an enhydris database we should *ALWAYS* exclude the UserProfile since we don't want user specific data to be transfered along with the rest of the database.

When adding a cronjob, if you don't want a regural mail to come after every sync, you should use the `--silent` option which redirects `stdout` to `/dev/null` and only prints `stderr`. This, coupled with the `-W` python flag can be used to make a cronjob send an email only whenever a problem was encountered. A sample cronjob which runs every night would be something like this:

```
1 0 * * * /usr/bin/python -Wignore manage.py hcore_remotesyncdb -a hcore -r itia.hydroscope.gr -e Use
```

How stuff works

In this section, we'll analyze the replication script and see how it operates behind the scenes. Of course, if you want to understand how it works it's probably better if you looked directly into its source code. Regarding the API which provides us with the database objects, it's been fully documented *here*. Here, we'll see how the replication script handles that data and adds it in the local database.

One important thing that you should be familiar with before we delve into the code is the difficulties that we came

across when trying to implement this feature. Postgres (and most databases by design) keep track of foreign keys using the primary key of an object which in all of enhydris models happens to be the object id. Since we want to aggregate multiple instances into one, it's only natural that there will be id collissions should we try to load the objects in the database while keeping their original id. Thus, we decided that keeping the ids intact was not an option and we had to find a way to preserve foreign keys and many to many relations without counting on object ids.

The best workaround is to add the objects without their foreign keys and many to many relationships and once the objects are in the database we could reinitialize all object relationships. To do that, we added two extra fields on all top-level objects named `original_id` and `original_db` which can be used to identify a specific object during the syncing process given that we know its id and the database that we're pulling the data from. Now the only thing was to somehow store the foreign relations in a way that could be parsed easily and quite fast after the object initialization. This was achieved using a multilevel dictionary which stores all object foreign relations and parsing this would be a breeze using python's optimized dictionary parsing routines.

Of course, that's when the real problems surfaced. Many objects have `Null=false` in some foreign keys which caused the replication to fail when trying to save objects with null foreign keys. In order to circumvent that, when firing up the replication script we create a set of `Dummy Objects` aka objects that have null values and are used to fill-in the not-Null foreign key dependencies of the to-be-installed objects. Once the replication objects are into the database, we delete the Dummy Objects and update the foreign relations to the original ones which we have stored in the dictionary mentioned above. This may be a slow process but is the only feasible solution that we came up with at the time.

Having said all that, we can see what the workflow of the script looks like. First of all, given the application name, it tries to import the specified app and list all available models in it. Using a multipass bubblesort algorithm, it sorts all models using their dependencies as specified in the `f_dependencies` model field and given that there are no circular dependencies, the final list contains the models in the correct replication order.

Using the model list, the script asks from the remote instance the JSON fixture of each model in the list which is fetched and saved in a temporary dir (by default this is `/tmp`). Once all JSON fixtures have been fetched, the script creates the generic objects and then deserializes each JSON file in the same order it was fetched. For each object within the fixture, it first strips all foreign relations and reinitializes the not-null ones using the generic objects. Also, the fields original_id and original_db are filled in and the foreign keys and many to many relations are saved in a multilevel dictionary for future reference.

Once the deserialization of all fixtures has been completed, all objects are saved under the same transaction management because we don't want to have any objects left out from the replication routine. If everything has been completed successfully, the script reinitializes all foreign keys and many to many relations from the dictionary and exits after cleaning up. If a problem occurs all transactions are rolled back and the database is exactly as it was before the replication attempt.

Note:

The generic objects which are used to fill temporary *Not Null* foreign relations are handcrafted. This means that should the Enhydris database schema change drastically, this would probably require an update as well.

CHAPTER 7

permissions — Permissions

This module implements row level permission handling to use along with django's generic permissions provided by the django.contrib.auth module. More precissely, this module extends the User and Group models with a couple of methods which take care of adding,deleting and checking of permissions. The *Permission* class keeps log of all existing permissions in the database.

7.1 Permission Objects

Each instance of the *Permission* class represents a relationship between a user and an object and it is identified by its name. The permission name can be any string like 'edit', 'read' or 'delete' and usually describes the kind of permission it implements.

class permissions.**Permission**(*name*, *content_type*, *object_id*, *content_object*[, *User*, *Group*])

name
The name of the permission. Usually it's a string denoting the meaning of the permission (eg 'edit', 'read', 'delete', etc)

content_type
This attribute stores the content type of the object over which this permission is effective.

object_id
This is the id of the related object.

content_object
This is a foreign key to the actual object (object instance) over this permission is effective.

user
If the permission is effective for a single user, this field points to this user otherwise it is null.

group
If the permission is effective for a whole group, this field points to this group otherwise it is null.

7.2 User/Group methods

As told before, the row level permissions add various methods to the User and Group models with which one can add/edit/delete permissions over various objects and/or QuerySets.

class User:

permissions.**add_row_perm**(*instance*, *perm*)

This method takes an object instance and the name of the permission and adds this permission for the calling user over the object instance given. For example:

```
>>> station = Station.objects.get(id='10001')
>>> user = User.objects.get(username='testuser')
>>> user.add_row_perm(station, 'edit')
```

permissions.**del_row_perm**(*instance*, *perm*)

This method takes an object instance and a permission name and if the user has that permission over the object, the method deletes it. If the user doesn't have that permisssion, nothing happens.

```
>>> station = Station.objects.get(id='10001')
>>> user = User.objects.get(username='testuser')
>>> user.del_row_perm(station, 'edit')
```

permissions.**has_row_perm**(*instance*, *perm*)

This method takes an object instance and a permission name and checks whether the calling user has that permission over the object instance. If this method is called from a superuser, it always returns `True`. For example:

```
>>> station = Station.objects.get(id='10001')
>>> user = User.objects.get(username='testuser')
>>> user.has_row_perm(station, 'edit')
False
```

permissions.**get_rows_with_permission**(*instance*, *perm*)

This method is used to return all instances of the same conten type as the given instance over which the user has the *perm* permission. For example:

```
>>> user = User.objects.get(username='testuser')
>>> user.get_rows_with_permission(Station,'edit')
```

This will return all Stations that the user can 'edit'.

class Group:

All methods and their usage are the same as with User. However, it's worth noting that once a user inherits a permission from a group, the only way to remove that permission is to leave the group since using *del_row_perm()* from the user won't affect the group permissions.

permissions.**add_row_perm**(*instance*, *perm*)

permissions.**del_row_perm**(*instance*, *perm*)

permissions.**has_row_perm**(*instance*, *perm*)

permissions.**get_rows_with_permission**(*instance*, *perm*)

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

d

p

A

C

D

E

S

T

U

V

W