HappyFace Documentation

Release 3.0 RC1

Gregor Vollmer, Stefen Roecker, Marcus Schmitt, Stefan Wayand

July 18, 2013

CONTENTS

- Powerful site specific monitoring system for data from multiple input sources.
- Collects, processes, rates and presents all important monitoring information
- for the overall status and the services of a local or Grid computing site.
- Monitoring data is subdivided in multiple categories.
- Each category is subdivided in multiple modules which corresponds to one single test.
- Simple rating system: -1 = no info / error; status float value = 0.0 .. 1.0 (critical .. fine)
- The overall status of the categories can be calculated from the individual module statii with different algorithms.

CHAPTER

ONE

BASIC OPERATION AND CONCEPTS

HappyFace is a flexible meta-monitoring system, designed to aggregate expert monitoring data from various sources. It gives shifters and site maintainers the ability to check the site status by looking at a single web page. The data is not only aggregated and displayed on web page, but also stored in a database, giving shifters and administrators a powerful tool to introspect the site status at any time in the past.

1.1 HappyFace Workflow

1.2 Description of HappyFace Parts

1.2.1 HappyFace Core

1.2.2 Support Tools

1.2.3 Worker Modules

1.2.4 Weboutput

1.2.5 Plot Generator

CHAPTER

TWO

INSTALLATION

Please read *Basic Operation and Concepts* before trying to install HappyFace, as it might save you from some headaches.

2.1 Dependencies

HappyFace currently requires the following Python dependencies (Debian package names)

- python (version >= 2.6)
- python-cherrypy3
- python-sqlalchemy (version >= 0.5)
- python-migrate (for *dbupdate*-tool)
- python-mako

The following are some optional packages

- python-sqlite
- python-psycopg2
- python-matplotlib
- python-lxml
- more database adaptors

2.2 Getting the Source

The simplest way is to checkout the current source code from the HappyFace repository

```
$ svn co https://ekptrac.physik.uni-karlsruhe.de/public/HappyFace/branches/v3.0 HappyFace
```

At this point, there are no modules available, you have to get them separately from any source you like. For example, checkout the Karlsruhe modules

```
$ cd HappyFace
$ svn co https://ekptrac.physik.uni-karlsruhe.de/public/HappyFaceModules/trunk modules
```

2.3 Configuration

Since the configuration in HappyFace is rather flexible, the *next chapter* is devoted to the configuration of HappyFace.

For testing purposes, you can download an example configuration and extract it into a subdirectory called config/ in the HappyFace directory.

2.4 Running HappyFace in an Development Environment

You should have a configured copy of HappyFace by now. Let the path to it be */path/to/HappyFace*.

You can now manually run

```
python acquire.py
```

to populate the database. By running

```
python render.py
```

you start a local webserver, by default listening to port 8080, so you can access your instance at http://localhost:8080/. If you change the Python source files, the server process automatically reloads itself, so you can keep it running in a separate shell all the time.

Note: If you make syntax errors while programming, the server shuts down completely and you need to manually restart it.

The development server can, if properly configured be used to drive the site, too, since CherryPy claims the server to be powerful and reliable enough. But we advice to use Apache in a production environment, because it has better SSL support. Particularily, SSL certificate authorization is **not** possible with the built-in server.

2.5 Setting up HappyFace with Apache2 and mod_wsgi

You should have a configured copy of HappyFace by now and an installed an Apache web server. Let the path to it be */path/to/HappyFace* and let it belong to a user called *hfuser*. We want the HappyFace instance to be mounted at / of the URL path.

To run HappyFace with Apache, we advice you to use mod_wsgi, so make sure it is installed and enabled in your Apache server. See the Apache documentation if you need help with that.

You have to tell WSGI where the *render.py* script of HappyFace is located, as well as the URL where to mount it with the *WSGIScriptAlias* directive.

The Apache process needs to be restarted if the source code or configuration of HappyFace changed, otherwise changes take not effect. Because this usually requires root privileges, any user in the position to update HappyFace would also need root privileges. This is undesirable in most environments, so you should separate the Python process.

To do this, the *WSGIDaemonProcess* is used to spawn new processes in a process group. A single process group is usually okay for multiple HF instances. For every virtual host with a WSGIScriptAlias specification, you have to tell Apache to separate the processes with the *WSGIProcessGroup* directive.

The *WSGIScriptAlias* and *WSGIDaemonProcess* have many options that may be of use. Consult the mod_wsgi documentation for a full overview of their options.

An example configuration looks something like this

```
<VirtualHost *:80>
        ServerAdmin admin@example.com
        ServerName happyface.example.com:80

        <Directory />
                Order deny,allow
                Deny from all
        </Directory>

        WSGIScriptAlias / /path/to/HappyFace/render.py

        ## OPTIONAL: have HappyFace run in a separate process belonging to the HappyFace user
        WSGIDaemonProcess happyface user=hfuser
        WSGIProcessGroup  happyface

</VirtualHost>
```

2.6 Using PostgreSQL

By default, HappyFace uses a file-based "SQLite <sqlite.org"_ database to store all its stuff. While this is quite powerful and scales to some degree, a PostgreSQL database is more roboust when it comes to server crashes and backups.

In this chapter, we will setup HappyFace to use a PostgreSQL database we create before. It applies to other database systems as-well. The given code examples should work on Ubuntu, but won't differ very much from any other distro.

2.6.1 Install PostgreSQL

Using the package manager of your system, install the PostgreSQL database server and make sure it is startet.

```
$> apt-get install postgresql
$> /etc/init.d/postgresql status
Running.
```

If you plan to run your database on a server different to your HappyFace machine, you have to make sure the server listens to your network interfaces

2.6.2 Setting up a database

Most database operations can be performed with commandline commands. To use them, open a shell as your PostgreSQL user. Because he does not have a password (by default), log in via the root account.

```
    $> su
Password:
#> su postgres
$>
```

Before we create the database, we create the user (here: *happyface*) that we use to access the database and set his password. On the commandline create a user with the *createuser* command.

```
# ask for password when creating user
$> createuser -P happyface
Enter the password of the new role:
Enter it again:
```

Now we can create a database (here: *my_hf_instance*) and set our HappyFace user as owner

```
$> createdb my_hf_instance -O happyface
```

2.6.3 Configuring HappyFace

The configuration process of HappyFace is explained in detail in the *next chapter*, so here only minimal steps are given.

Create a file *database.cfg* with the following content

```
[database]
url = postgres://happyface:PASSWORD@localhost/my_hf_instance
```

2.7 Certificate Authorization with Apache2

HappyFace can be configured to restrict access on certain modules to a small group of users. These users can identify themselves with a client certificate. For this to work, both HappyFace as well as Apache2 need special configuration.

Note: Certificate authorization does **not** work with the development server.

The Apache configuration for HappyFace needs to be duplicated for both the plain text HTTP as well as encrypted HTTPS configuration. To avoid code duplication, you should put the configuration inside the *VirtualHost* blocks into a separate file that is included with the *Include* statement.

2.7.1 Apache Configuration

We have to tell Apache2 to use SSL and client certificates, first. We assume you already have SSL certificates for your server as-well as the root certificate of the users you want to accept.

The root certificate(s) is/are the first line of authentification, the client certificate must match the given root certificates, otherwise access is automatically forbidden.

```
NameVirtualHost *:443
<VirtualHost *:443>
        ServerAdmin admin@example.com
        ServerName happyface.example.com:80

        SSLEngine On
        # Replace these paths with your own certificates
        SSLCertificateFile    /etc/apache2/server.crt
        SSLCertificateKeyFile /etc/apache2/server.key
        SSLCACertificateFile  /etc/apache2/gridka-root-cert.crt # alt.: SSLCACertificateDirectory

        SSLOptions      StdEnvVars
        SSLVerifyClient Optional  # Optional or Require

#       [...] Place usual HF config here
</VirtualHost>
```

The *SSLOptions* tells Apache to pass the required SSL informations to HappyFace. The *SSLVerifyClient* directive switches on client verification. Two reasonable settings are *optional*, which allows users without certificate to use SSL to access the site, and *require*, which has broader browsers support.

2.7.2 HappyFace Configuration

Apache now asks a client to show a certificate and checks if it is valid. We need to tell HappyFace which distinguished names (DN) are valid and to which categories and modules the access is restricted.

Because we only cover installation, deployment and Apache configuration in this chapter, we ask you to refer to *Certificate Authorization* for detailed information.

CHAPTER

THREE

CONFIGURATION AND SITE MAINTENANCE

The HappyFace 3 configuration is very flexible and can be bent to suit almost all needs in deployment and usage. In contrast, its predecessor HappyFace 2 was limited to a mildly confusing configuration style.

3.1 Core Configuration

Several aspects of HappyFace can be tuned by configuration, but usually you don't need extensive customisation, therefore meaningful defaults are required. To accomplish this, the configuration of the core takes place in two stages and places.

defaultconfig/ As the name suggests, HappyFace evaluates this directory first, gathering the configuration from all files ending in **.cfg* in alphabetical order. In a fresh checkout, there is only one such file, it contains all sections and options that are interpreted by HappyFace with their defaults. If you want to alter anything, copy&paste from there into a user configuration is a good start.

You should not alter the configuration in this directory since it is under revision control and might be altered in future! The only thing you are supposed to do with this directory is adding a new, unrevisioned file like this

```
[paths]
local_happyface_cfg_dir = /desired/path/to/configuration
```

This will change the path of the next configuration stage, where your user configuration is saved. You might want this if you wish to move configuration to places like */etc*, but is not required for a default setup.

Note: For the changes to be applied, make sure the name of the additional file is alphabetically after *happyface.cfg*!

---

config/ As stated in the previous section, the place of this directory can be changed, by default it is right inside the HappyFace directory.

Again, all files ending in **.cfg* are interpeted in alphabetical order, files occuring "later" can override earlier statements. We advice adding files based on sections, or thematically grouped.

We found it useful in HappyFace2 to place the site configuration in a local Subversion repository, so any developer can easily get a development configuration that is identical to production and reinstalling a site after a potential crash would be easy. Because sensitive information, like database passwords, should not be checked into the repository, they can be easily placed in a separate, unrevisioned file (HappyFace2 was not able to do that!).

3.1.1 Sections

The following is a list of the different sections in the default configuration and their purpose and potential use. For an overview of all config variables, just look at the default configuration files. Everything should be sufficiently commented in-line.

[auth] Certificate authorization details, described in *Certificate Authorization*

[database] Everything in this section is passed to the sqlalchemy library. See the sqlalchemy docs for details.

[downloadService] You can set the timeout for downloads as well as options passed to all wget downloads. This is useful for adding certificates globaly in HappyFace.

[global], [/] and everything starting with / These are special sections, the values are Python statements, so strings need proper delimiters and lists are supported.

CherryPy is configured directly from these sections, so if you want to add custom sub-pages to HappyFace (without altering the dispatcher in some way) you can do it here!

[happyface] If you want to specify a custom order of categories (the default one is "pseudo random"), you can do this here.

[paths] The longest section, many paths and URLs are specified here. In multi-HappyFace setups you want to change the root URL given by *happyface_url*. If you want to put HappyFace in the URL *http://example.com/HappyFace/gridka*, you set

```
happyface_url = /HappyFace/gridka
```

Don't forget to change the path in the *WSGIScriptAlias* as in *Setting up HappyFace with Apache2 and mod_wsgi*.

[plotgenerator] If you wish to use the HappyFace plot generator, you have to enable it here. Additionally, you can specify the matplotlib backend to use your favourite.

[template] A few tweaks on the things that are displayed in the web output.

3.2 Logfiles

The Python logging module is used to generate log files. There are different configuration files possible for acquisition and rendering, by default they are the files ending in **.log* in the *defaultconfig/* directory. They are imported by the *logging* module, so please consult the Python documentation about their format.

3.3 Modules and Categories

No actual work is done by the HappyCore, all input processing and displaying is implemented in the modules. The required configuration consists of a mechanism to create and configure module instances and then group them into categories for displaying.

3.3.1 Directory Layout

You can override the locations by changing the **paths** section of the HappyFace configuration, but basically all category configuration is in *config/categories-enabled* and all module configuration in *config/modules-enabled*.

The reason for the *-enabled* suffix is that we originally thought about a Debian-style configuration, where the configuration is in *-available* directories and symbolic links to them in *-enabled* directories. We do not use this at the moment, but if you like to, feel free to do it that way!

As with other configuration directories, all files ending in *.cfg* are parsed in alphabetical order. Because of this, you can split up the different categories and modules over files as you wish. It is possible to have only one massive file for either modules and categories.

We advice you to use one file per category, with the same name as the category, and a similarily named file for the modules, containing all the configuration for all modules in that one category. This supports the original Debian-style configuration idea, but is also a nice grouping, since it is always clear where the module is located. This also makes the maintenance and setup of similar categories rather simple.

3.3.2 Create and Configure Module Instances

Each module instance has to be assigned a unique name that, it should only contain alphanumerical characters and underscores. We advice you to use a C-style **underscore_separated_name**, *not* CamelCase or something. The name is used at several places, internally, e.g. as anchor in HTML hyper links.

There are certain config variables common to all modules, as well as a set of variables dependent on the module type.

The easiest way to obtain a skeleton configuration is to use the *modconfig* tool. Just pass it the module type as a name and you get a skeleton you can easily adapt.

Common module configuration variables are

module The name of the module class that is used. If it does not correspond to one of the classes in the *modules/*, the world will be sucked into a cosmic singularity.

name A verbose name for the generated output

description

Todo

what is it?

---

instruction What should a shifter do if this module fails? Displayed in the module panel on the weboutput.

type How the module will affect category status calculations. Possible values are

rated Uses status mechanism and is taken into account when calculating the status of a rated category

unrated A status is calculated and displayed for the module, but it is ignored when calculating the category status.

plots The status only encodes "got data" and "got no data", taken into account by plots categories.

weight A numerical value that should be between 0.0 and 1.0 that might be taken into account by some algorithms to calculate the category status.

Example

As an example, consider we want to configure an instance of the *Plot* module, downloading an image from *https://example.com/file.png*. To obtain a basic configuration, we type the following command on the command line

```
python ./tools.py modconfig Plot
```

and are rewarded with the following output

```
[INSTANCE_NAME]
module = Plot
name =
```

```
description =
instruction =
type = rated
weight = 1.0


# Enable the mechanism to include two timestamps in the GET part of the URL
use_start_end_time = False

# Name of the GET argument for the end timestamp, which is now
endtime_parameter_name = endtime

# URL of the image to display
plot_url =

# Name of the GET argument for the starting timestamp
starttime_parameter_name = starttime

# How far in the past is the start timestamp (in seconds)
timerange_seconds = 259200
```

All we have to do now is replace the section name, change the *type* to *plots*, set a verbose name and insert a valid download command for the *plot_url* variable.

3.3.3 Configuring Categories

Each category corresponds to a page on the HappyFace weboutput and is a logical group of HappyFace modules. Modules are specified by creating a uniquely named section in a **.cfg* file in the category config directory.

Configuration Variables

The *CATEGORY_ID*, the name of the config file section, is a short version of the name that should only constist of alphanumerical characters and underscores, since it is part of the URL pointing to the web page. The available variables inside the sections are

name The verbose name of the category

description A short description of the category that can be displayed somewhere (not used in default templates, at the moment).

type Choose if category is informational only or has a status value. Set to one of the following

plots No status is calculated for this module, since only informational or not parsable data (e.g. images) are contained.

rated Calculate a module status with the specified *algorithm* and display the result on the webpage accordingly.

algorithm The algorithm to calculate the category status with. At the moment, *worst* and *average* are available.

Setting the Order of Categories

Usually, you want to display the categories in a certain order on the webpage. For this reason, there is the **categories** variable in the *[happyface]* section of the core configuration. Just enter a colon-separated list of categories there and they will be included on the weboutput. If you do not specify **categories**, the categories will appear in arbitrary order.

3.4 Updating the Site

Basically, the update flow is as follows

1. update the core
2. update the modules
3. restart the server (e.g. *pkill apache2* as unprivileged HappyFace user with *mod_wsgi* configured accordingly)
4. run *python tools.py dbupdate -f*

Note: You probably get warning messages when using *pkill* to restart your server process(es), because you might try to kill processes from other users. Just ignore those messages.

Note: The process name in newer versions of Apache seems to be *apach*, so to restart you need to call *pkill apach*.

Updating the source code of HappyFace or its modules might render the database schema incompatible. In this case, HappyFace tries to throw supportive error messages, giving you hints that schema updates might be neccessary.

To update the database schema, the *dbupdate* tool is used. To check if updates are neccessary, you can do a dry run. This will list all required changes.

```
python tools.py dbupdate --dry
```

Note: With some database backends, *dbupdate* falsely claims that altering some columns is neccessary. This is unfortunate, but should not cause any problems. Because of this a run through *dbupdate* might take some time, although basically nothing happens.

We hope to resolve this issue soon in an updated version of the *dbupdate* tool.

If you see that updates are required, you can either run it interactively, without any command line parameters, or trust that all changes are valid and force the update by

```
python tools.py dbupdate --force
```

3.5 Certificate Authorization

In the section *Apache Configuration* we covered the neccessary configuration for the Apache server to support client certificates. We tie in with this and cover the configuration of HappyFace.

HappyFace gets a certificate DN from Apache but still has to decide if access is granted to that particular user. And secondly, we need to tell HappyFace which modules and categories need to be secured, after all.

3.5.1 HappyFace Configuration

After the client certificate is verified (it matches the given root certificates), HappyFace checks if its DN is authorized to access the secured parts of HappyFace.

Two mechanisms are available for this in HappyFace, which can be configured in the [auth] section of the HappyFace configuration. From the default configuration we have

```
[auth]
# A file containing authorized DNs to access the site.
# One DN per line
dn_file =

# If the given DN is not found in the file above, if any, the following
# script is called with DN as first argument.
# The script must return 1 if user has access, 0 otherwise.
auth_script =
```

At first, the DN from the client is checked against a list of DNs in a file. This is a quick check and since it is a plaintext file, it is easy to maintain.

The second mechanism allows the use of other data sources. A given executable is run, doing whatever it wants and exit with a certain status code. If the status code is 1, the user may access the secured parts of HappyFace, if the status is 0 (or anything != 1), the client may only access the public parts of HappyFace.

Note: The script is run whenever a URL below the HappyFace root url is accessed.

If the authorization script needs to perform expensive or time consuming operations (complex DB queries or slow web queries), you should cache the results locally. For this, a simple SQLite Database should be sufficient.

An example utilizing a Python script to query sitedb. Anyone with a CMS account (technically, where the DN can be converted to a CMS account) has access. Since the script is very simple, it does not cache the results, therefore accessing HappyFace is slowed down noticeably.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

import os, sys, httplib, json, urllib, ast

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print "Single Argument, DN, required"
        sys.exit(0)

    con = httplib.HTTPSConnection('cmsweb.cern.ch', 443)
    con.connect()
    try:
        dn = urllib.quote_plus(sys.argv[1])
        con.request('GET', '/sitedb/json/index/dnUserName?dn='+dn)
        response = con.getresponse().read()
        response = ast.literal_eval(response)
        print "Authorized DN"
        sys.exit(1)
    except SyntaxError:
        # Literal eval failed, crazy exception "JSON"
        print "Unknown DN"
        sys.exit(0)
    finally:
        con.close()
```

3.5.2 Module and Category Configuration

Limiting access to a module or complete category is very simple. In both the module and category configuration files, the *access* option is supported.

For modules, it must be set to *restricted* or *open*. By default, if *access* is not specified, *open* is assumed. These may be overridden by the category access.

Categories accept *restricted*, *permod* and *open* as valid values for *access*. Basically, *open* and *restricted* set the access option of all included module to the corresponding value, making either all modules open or requiring authorization for all of them.

If you only want to restrict some modules, use *permod*, which is the default. Only in this case the module access variable is used.

> **Warning:** Be careful when using *open* together with categories, as you might involuntarily expose sensitive information.

For completeness, an example category configuration is given where the access to all modules restricted.

```
[batch_system]
name = Batch System
algorithm = worst
type = rated
description =
modules = gridka_jobs_statistics

access = restricted
```

3.6 Hints on Databases and Optimization

Since HappyFace stores alot of data in standard database systems, delivery time and performance of HappyFace depends on data retrieval. There are things that should be taken into account when choosing and configuring a database for HappyFace.

We are no database experts, so this section is mostly our experiences with HappyFace3 and the database systems we used and tested, this is by far not a complete overview of all possible databases, optimization variables or pitfalls. If you encounter issues that should be added to this guide, feel free to contact our development team.

3.6.1 SQLite

The database of choice for development environments is SQLite, mainly because no additional configuration is required. Taking backups of SQLite databases, besides copying the database file, is not sanely possible, so this is one reason not to use it in an production environment. To be honest, though, it can handle surprisingly large (~ 20GB) database files.

You might encounter situations where the database is locked, causing all database transactions with a **Database Locked** exception. We do not know where this comes from, probably simultaneous access across processes. So far it only occured on development machines, but with both HappyFace2 and HappyFace3. The common solutions found on the internet to fix the databae do not work for some reason, so your only option would be removing the database.

3.6.2 PostgreSQL

sqlalchemy uses connection pooling to speed up sequential database access. For some reason, we had occasional performance issues after short, many HappyFace accesses, that were tracked to the database and idleing processes. It was possible to recover the usual page access times by restarting either PostgreSQL or the HappyFace process, so we assume the error had something to do with the database.

A permanent solution to fix this was setting the *recycle* option in *sqlalchemy*, so our database configuration looks like

```
[database]
url = postgres://USER@SERVER/DATABASE
recycle = 10
```

Basically, this closes used connections after 10 seconds of inactivity and opens a fresh connection instead. We did not have these performance issues after wards.

CHAPTER

FOUR

MODULE DEVELOPMENT

Modules are the building blocks that give HappyFace its functionality. Specifically, they

1. download and parse data to store it in a database
2. retrieve data from a database and render it to HTML

These actions are represented by the `render.py` and `acquire.py` entry scripts. In order to do their work, they need to interact with the core of HappyFace and have to obey the rules for modules, otherwise the functionality of external tools and HappyFace itself can be disturbed.

4.1 Module Basics

The module source code is located somewhere within the modules directory of the HappyFace setup, by default it is just the modules/ subdirectory. At startup, everything inside this directory and any subdirectory that looks like a python module is imported and module wide source code is executed.

In one of these files is the source code of our example module. The module itself is a class derived from `hf.module.ModuleBase` that must implement a certain subset of methods, as well as specify a set of class-level variables describing configuration and database layout

Additionally, a HTML template file with the same name as the class is expected along with the file the class is in. This template contains the formatting information for the web output.

4.1.1 Database Layout

Associated with each module is a so called module table in the database and an arbitrary number of subtables.

One entry is added to the module table every time `acquire.py` is called. By default, it contains the only a minimal set of columns, but can be extended with the `table_columns` variable in the module definition. For the module developer, the following columns are of interest

term status

A numerical value. Its meaning is as follows

- 0.66 <= status <= 1.0 The module is happy/normal operation
- 0.33 <= status < 0.66 Neutral, there are things going wrong slightly.
- 0.0 <= status < 0.33 Unhappy, something is very wrong with the monitored modules
- status = -1 An error occured during module execution
- status = -2 Data could not be retrieved (download failed etc.)

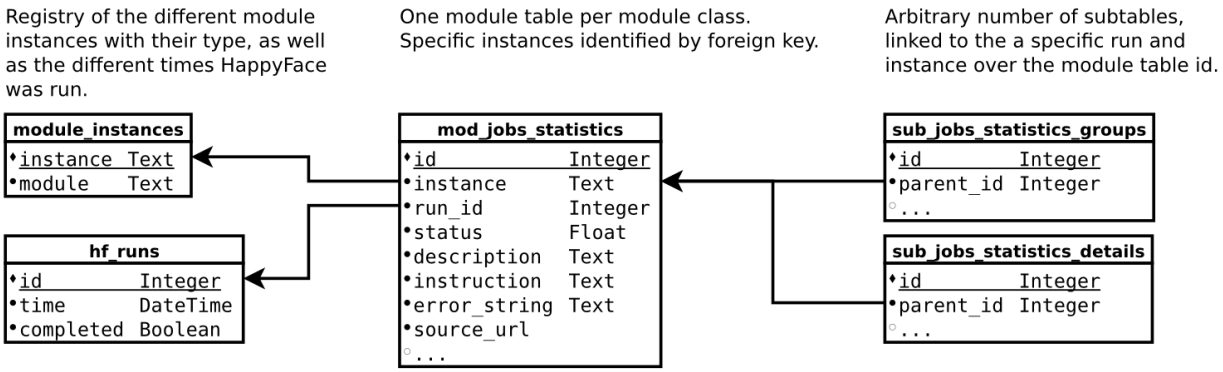
next term source_url An URL to the data source, if applicable. At the moment only a single URL can be specified, this is to be regarded as a current limitation of HappyFace.

Subtable System

Often you want to store more data than just a single, fixed entry in the database per run. For this the so called subtable mechanism is used. For each module, you can specify an arbitrary number of subtables. Each entry in them is linked to an entry in the module table. By this, the data is in the subtable is uniquely identified to a point in time and a specific module instance.

HappyFace cannot provide you much help for filling the subtables and inserting the data into the template namespace. For that reason you have to use the sqlachemy API to insert and select the data.

The following graph is a schematic overview over the relations between the main HappyFace database registry, the module tables and subtables. Only the columns common for all module and subtables are shown. In reality, there are user defined columns for module and subtables containing the actual data.



Smart Database Filling

Sometimes you encounter situations where you want a module to store *a lot* of information that only changes once in a while. For example, we weekly generate an overview of the stored files in our computing site. Obviously, this is a lot of data you don't want to store quarter hourly.

To avoid rewriting detailed data in subtables, the smart filling mechanism can be used. It is an easy way to tell HappyFace whether or not subtable data shall be stored. In order to correctly display the module, one has the keep track when detailed information was stored last.

In order to use smart filling, set the *class-level variable* **:dat:‘use_smart_filling‘** to *True*.

```
class MyModule(hf.module.ModuleBase):
    use_smart_filling = True
    # [...]
```

Internaly, HappyFace will then add another column named *sf_data_id* to the module table, pointing to the actual data, but this should be of no direct concern for module developers.

When smart filling is used, the two attributes `smart_filling_current_dataset` and `smart_filling_keep_data` are available.

4.1.2 Example

A minimalistic, working example of a Python module is presented here

```
# Module Definition
import hf
from sqlalchemy import *

class Dummy(hf.module.ModuleBase):
    config_keys = {'test': ('A config variable that is directly passed into the database', '')}
    table_columns = [Column('test',  INT)], []

    def extractData(self):
        return {
            "status": '',
            "test": int(self.config['test'])
        }

## HTML Template
<%inherit file="/module_base.html" />

<%def name="content()">
<p>${dataset['test']}</p>
</%def>
```

A detailed description of the module class variables and methods are found in the next section. The Mako Templating Engine is used for parsing the HTML template, please consult the Mako Documentation for more information about the syntax.

4.2 Module Class Reference

The module class is derived from `hf.module.ModuleBase` and the naming should be CamelCased. For the database table names, the CamelCase name is converted to camel_case.

Any class defiving from `hf.module.ModuleBase` found in the modules directory somewhere is considered a HappyFace module. It is then checked if

4.2.1 Special Class Variables

HappyFace makes use of class wide variables to define several aspects of the module.

config_keys
required

A dictionary where the keys correspond to module specific keys in the configuration file and the value is a tuple of two strings. The first string is a description of the variable and the second one a string with the default value (e.g. empty string).

This is used by the `hf.tools.modconfig` to generate empty configurations for a module.

config_hint
optional

A plain string with general information about the configuration of the module. Used by `hf.tools.modconfig` where it is put at the top of the automatically generated configuration, if specified.

table_columns
required

A tuple with two lists in it. 1) A list of sqlalchemy Column objects. These columns are added to the module table and usually suffice for the module operation 2) A list of strings, they are the names of columns in the module table that point to files in the archive directory.

subtable_columns
optional

A dictionary where the key is the name of the subtable, e.g. *details*, and the values are tuples like `table_colums`. They are the data columns for the subtable and the corresponding archive links. For more information about subtables, see *Subtable System*

The subtable names are not passed to the database as they are, but are prepended with the module name to ensure uniqueness. Therefore, two modules can use the same subtable name without problems.

The resulting Table objects can be accessed via the `hf.module.ModuleBase.subtables` dictionary.

use_smart_filling
optional

Set to *True* if you want to enable *smart filling* on the module.

4.2.2 Class Methods

`hf.module.ModuleBase` does provide several convenience functions that are used when the HTML weboutput is created, as well as default implementations for some optional actions the module can perform. The functions are called during different steps of the HappyFace acquire and render run and perform specific actions.

In total, you must implement at least one method, `hf.module.ModuleBase.extractData()`, to populate the database and optionally, a set of the following methods

- `prepareAcquisition()`
- `fillSubtables()`
- `getTemplateData()`

Please refer to the linked documentation of `hf.module.ModuleBase` and the *Step-by-Step Guide*. for implementation details

4.3 HTML Templates, Generating Output

By now we acquired data, stored them into the database and maybe wrote a function to retrieve data from the database again. To actually display something on the HappyFace weboutput, you need to create an HTML template first.

Internally, the Mako Template Engine is used interpolate the data into the template.

In short, all data is filled in the template with a pattern looking like *${* **expression** *}*, where expression is a piece of Python code returning a value. By default, the expression is converted to a unicode string, if it isn't already, and then HTML special character escaping is performed. This results e.g. in the replacement of **&** to **&**. If you do not want these default filters applied, to can disable them with the *n* filter as described in the documentation. In that case, you need to take care of applying the filters yourself.

Useful data is stored in template wide variables. By default, the following variables are available, although you usually only need few of them.

hf The HappyFace namespace, refer to the *Core documentation* for an overview. Usually, you only need functions from `hf.utility` or `hf.url`.

module The module object of the current instance.

data_stale A flag that indicates if the data is stale, meaning it is older than a certain time threshold.

run The run dictionary with the information about the current run.

dataset Often the only variable you need. It is the data in the module table from the current run. This is the data you probably want to display.

The template namespace is extended by the dictionary returned by `getTemplateData()`. So if you return a dictionary with the key *super_special_data*, the variable in the template with the same name has the value of the key in the dictionary.

Todo

Include information about CSS and JavaScript.

4.4 Using Matplotlib

Sometimes you don't want to display raw, numerical data but instead generate a nice plot from your data. Matplotlib is the common choise for this in Python and is used by the internal plot generator.

If you want to use Matplotlib in one of your modules, you must not include *pyplot* on the module level, but only in the class methods where you want to use it. If you don't do that, you will get a warning message as the following when running the server.

```
/usr/lib/pymodules/python2.6/matplotlib/__init__.py:856: UserWarning:  This call to matplotlib.use()
because the the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

if warn: warnings.warn(_use_error_msg)
```

The reason for that is that HappyFace tries to set the backend during the configuration phase as it is requested by your configuration files. If you import *pyplot* at the module level, it is imported way before the backend is set, which subsequently fails.

4.5 Step-by-Step Guide

Todo

write step-by-step module development guide

CHAPTER

FIVE

DYNAMIC PLOT GENERATOR

An important desing concept of HappyFace is saving a long-term history of monitoring data, so that events (e.g. software failure or dead-locks) can be analyzed for time correlation with other events. Instead of looking on tables of data, looking at a visual representation often yields a better understanding. As an example, the number of cache transfers increses at the same time as the response time decreases.

Because looking at the raw data in image form is a recurring requirement in HappyFace, it provides the so called plotgenerator for generating graphs. It cannot only be used by modules to generate images, but with a comfortable web interface site users can generate plots on their own.

5.1 Column expression layout

Test

CHAPTER

SIX

CORE DOCUMENTATION

HappyFace implements many classes and functions for internal use or for external tools.

Note: Many of these classes are of **limited to no use** for the average **module developer and site maintainer**. If you want to get started with module development, please consult the module development guide. It links to this reference where the module developer needs to interact with core classes.

6.1 hf.auth – Certificate Authorization

hf.auth.**cert_auth**()

hf.auth.**init**()

6.2 hf.category – Category Management and Weboutput

hf.category.**config**

class hf.category.**Category**(*category_name*, *conf*, *module_list*, *run*, *template*)

For the meaning of status values, see ModuleBase docstring.

getIndexIcon()

getLockIcon()

getStatusIcon()

hasUnauthorizedModules()

isAccessRestricted()

isUnauthorized()

render(*template_context*)

url(**args*, ***kwargs*)

class hf.category.**CategoryProxy**(*name*, *conf*, *module_conf*)

A run independant Category object. Can create run dependant category objects efficently.

For the meaning of status values, see ModuleBase docstring.

acquire(*run*)
Acquire data and fill tables for a certain run.

The ModuleProxy takes care that the call is "independant", so we do not need to care here =)

getCategory(*run*)

hasUnauthorizedModules()

isAccessRestricted()

isUnauthorized()

prepareAcquisition(*run*)
Prepare data acquisition for a certain run.

The ModuleProxy takes care that the call is "independant", so we do not need to care here =)

template = **None**

class hf.category.**Dispatcher**(*category_list*)
Show a page for displaying the contents of a category.

default(*category=None*, ***kwargs*)

prepareDisplay(*category=None*, ***kwargs*)
generate the data and template context required to display a page containing the navbar.

Parameters **string** – Name of the ca

Returns tuple (template_context, category_dict, run)

hf.category.**renderXmlOverview**(*run*, *template_context*)
Create a summary of the status of all categories and their modules in an XML format.

Useful for the HappyFace Firefox Icon or the HappyFace AndroidApp.

6.3 hf.configtools – Config Parser and Startup

class hf.configtools.**ConfigDict**

hf.configtools.**readConfigurationAndEnv**()

hf.configtools.**setupLogging**(*logging_cfg*)
Setup the python logging module and apply loglevel from environment variables if specified.

The argument to this function is the name of the configuration key in the 'paths' section containing the path to the loggin configuration. For the format of the config, please see the Python docs http://docs.python.org/library/logging.config.html#configuration-file-format

If None is specified, console logging only is setup. This is particularily useful for tools e.g. for migration or cleanup.

The environment variable HF_LOGLEVEL can contain a level from DEFAULT, INFO, WARNING, ERROR or CRITICAL. If HF_DEBUG is set, HF_LOGLEVEL=DEBUG is implied.

6.4 hf.database – Database Setup

hf.database.**connect**(*implicit_execution=False*)

hf.database.**disconnect**()

6.5 hf.dispatcher – Root CherryPy Dispatcher

class hf.dispatcher.**CachegrindHandler**(*next_handler*)
Callable which profiles the subsequent handlers and writes the results to disk.

Based on http://tools.cherrypy.org/wiki/Cachegrind

class hf.dispatcher.**RootDispatcher**
The main HF Dispatcher

archiveFileAuthorized(*filename*)

errorPage(***kwargs*)

index()

static(**args*)

hf.dispatcher.**cachegrind**()
A CherryPy 3 Tool for loading Profiling requests.

To enable the tool, just put something like this in a HappyFace configuration file:

```
[/category]
tools.cachegrind.on = True
```

This will enable profiling of the CherryPy code only for the category pages, not static content or the plot generator. As a result the performance impact is reduced and no files with uninteressting data are created.

6.6 hf.downloadservice – Downloads and Archive

class hf.downloadservice.**DownloadFile**(*download_command*)

copyToArchive(*module*, *name*)
Copy the file to the archive directory. The name is prefixed with the instance name of the module, thus the name should be always unique and the file can be associated with the module.

errorOccured()

getArchiveFilename()

getArchivePath()

getArchiveUrl()

getFile()

getSourceUrl()

getTmpPath(*no_exception=False*)

Parameters **no_exception** – default False. If set to True, no exception is thrown

when there was a problem while downloading the file.

isArchived()

isDownloaded()

class hf.downloadservice.**DownloadService**
Note when copying files to the archive directory: The name must start with the name of the module instance, otherwise certificate auth will always disable the file!

addDownload(*download_command*)

cleanup()

getArchivePath(*run*, *filename*)

getArchiveUrl(*run*, *filename*)

performDownloads(*runtime*)

class hf.downloadservice.**DownloadSlave**(*file*, *global_options*, *archive_dir*)

run()

class hf.downloadservice.**File**(*run*, *name*)

getArchiveFilename()

getArchivePath()

getArchiveUrl()

6.7 hf.module – Module Functionality

The very core of HappyFace are modules doing the actual work. All of them derive from ModuleBase, and any other

hf.module.**config**
A config parser instance with the aggregated data from all module configuration files. It is created and populated at initialization by hf.configtools.readConfigurationAndEnv().

Since this module variable is used by most category related methods, it is important to initialize it early.

class hf.module.**ModuleBase**(*instance_name*, *config*, *run*, *dataset*, *template*)
Base class for HappyFace modules. A module provides two core functions:

1.Acquisition of data through the methods

(a)prepareAcquisition: Specify the files to download

(b)extractData: Return a dictionary with data to fill into the database

(c)fillSubtables: to write the datasets for the modules subtables

2.Rendering the module by returning a template data dictionaryin method getTemplateData.

Because thread-safety is required for concurrent rendering, the module itself MUST NOT save its state during rendering. The modules functions are internally accessed by the ModuleProxy class.

The status of the module represents a quick overview over the current module status and fitness.

•0.66 <= status <= 1.0 The module is happy/normal operation

•0.33 <= status < 0.66 Neutral, there are things going wrong slightly.

•0.0 <= status < 0.33 Unhappy, something is very wrong with the monitored modules

•status = -1 An error occured during module execution

•status = -2 Data could not be retrieved (download failed etc.)

The category status is calculated with a user specified algorithm from the statuses of the modules in the category. If there is missing data or an error, the category index icon is changed, too.

In practice, there is no "visual" difference between status -1 and -2, but there might be in future.

Note: A lot of the configuration is done with class variables in the derived class. All special variables are listed *here*.

Internaly, they are used to generate the database *Table* objects, generate default configuration and set whether the *smart filling mechanism* shall be used.

It makes use of the `ModuleMeta` class internally.

module_table
Sqlalchemy *Table* object of the modules main data table

smart_filling_current_dataset
The data dictionary from the last run with data or *None* if the module never inserted data in the databae before.

Use *self.smart_filling_current_dataset["id"]* as parent ID for accessing data in subtables, since it belongs to the correct run containing data.

In `getTemplateData()`, it is equal to `dataset` if the run to display stored data.

See *Smart Database Filling* for more information,

smart_filling_keep_data
Boolean flag to indicate that data shall be stored for the current run. If it is set to *False*, `fillSubtables()` is not called.

Note: Attribute is ignored if set outside of `extractData()`.

Warning: The default on module creation is *None*, causing an exception to be thrown if you don't set the flag to *True* or *False* in `extractData()`!

See *Smart Database Filling* for more information,

subtables
A dictionary of sqlalchemy *Table* objects, with their given names as key.

module_name
The name of the module class.

Note: This is **not** the name of a specific instance!

---

instance_name
The name of the instance of the module currently processed.

config
A dictionary with the module configuration

run
Only available in `getTemplateData()`. A dictionary with the *id* and *time* of the current run to be displayed.

dataset
Only available in `getTemplateData()`. The data dictionary from the module table for the currently processed run.

category
A reference to the `hf.category.Category` object where the module is in.

template
The Mako *Template* object.

weigth
The 0..1 weight used in some category rating algorithms.

type
Either *plots*, *rated* or *unrated*.

extractData()
Mandatory function to process some data and return it in a format that can be used to populate the module table. Downloaded files, e.g. in XML format, should be parsed here.

If a part of the extracted data cannot be stored in the module table, but must be passed to a subtable, save it in a class variable. Then, save it into the databe with your own implementation of `fillSubtables()`.

For more information about subtables

Returns A dictionary where the names of module table columns and the values are the data to be inserted into the database. If a column is specified as a file column, objects with an *getArchiveFilename()* method are accepted. This is the case for `hf.downloadservice.DownloadFile`, returned by the download service.

ajaxUrl(**args*, ***kwargs*)

config_defaults = {'instruction': '', 'type': 'rated', 'description': '', 'weight': '1.0'}

filepath = None

fillSubtables(*module_entry_id*)
Override this method if your module uses subtables, to fill them with the data from `extractData()`.

To fill the subtable, you need the sqlalchemy table class and issue one or more insert statements. The Table classes are available in the `subtables` dictionary, where the key is the name specified in `subtable_columns`.

Note: This function is not called if smart filling is used and the current data does not need to be stored. Namely, `use_smart_filling` is *True* and `smart_filling_keep_data` is *False*.

For more information about subtables, see *Subtable System*

Parameters **module_entry_id** (*integer*) – The ID of the module table entry that works as parent to the subtable entries added by this call.

To fill a list of col->value dictionaries in an object attribute to a subtable called *details*, the following implementation can be used

```
def fillSubtables(self, module_entry_id)
    table = self.subtables['details']
    for entry in self.extra_data:
        table.insert().execute(dict(parent_id=module_entry_id, **entry))
```

Actually, the insert accepts a list of dictionaries, but because we do not have the parent ID in the dicts yet, we cannot add it directly.

An inline version using generator expressions would look like this

```
def fillSubtables(self, module_entry_id)
    self.subtables['details'].insert().execute([dict(parent_id=module_entry_id, **row) for r
```

Finally, a short, readable variant

```
def fillSubtables(self, module_entry_id)
    self.extra_data = map(lambda x: x['parent_id'] = module_entry_id, self.extra_data)
    self.subtables['details'].insert().execute(self.extra_data)
```

getAllColumnsWithSubtables()
Get the names of all columns for the module table and all subtables. The key of the module table is the empty string, the other keys are the names of the subtables.

Return type dict (subtable => list of columns)

getNavStatusIcon()
Get URL to a small status icon for the current module state.

This function uses the icons located in *path.template_icons_url*.

Note: This method only works in the render process.

---

getPlotableColumns()
Get the names of all columns than can be plotted, that is they are numerical and probably no IDs.

Return type list

getPlotableColumnsWithSubtables()
Get the names of all columns than can be plotted, that is they are numerical and probably no IDs, for the module table and all subtables.

The key of the module table is the empty string.

Return type dict (subtable => list of columns)

getStatusIcon()
Get URL to a large status icon for the current module state.

This function uses the icons located in *path.template_icons_url*.

Note: This method only works in the render process.

---

getStatusString()
Get a string describing the status of the module.

Depending of the module type the string is different, it can be used for example to get the name of an icon.

It is one of *noinfo*, *happy*, *neutral*, *unhappy*, *unavail_plot*, *avail_plot*.

Note: This method only works in the render process.

Return type string

getTemplateData()
Override this method if your template requires special preprocessing of data or you have data in subtables.

The `dataset` and `run` attributes are available in this method.

Returns A dictionary that extends the Mako template namespace.

isAccessRestricted()
Find out if access to the module is restricted.

Return type boolean

Returns

- True if a valid certificate is required to access the module
- False if access is open to anyone

isUnauthorized()
Check if the user from this request is authorized to access the module. This takes into account the module restriction and the user certificate, if available.

Note: This method only works in the render process.

Return type boolean

Returns

- True if the user must not access the module
- False if the user may access the module

prepareAcquisition()
Override this method if your module needs to download data, or has to perform some other action prior to the data acquisition run.

render()
Return a string with the rendered HTML module

url(*only_anchor=True*, *time=None*)
Get the URL to this module.

Parameters

- **only_anchor** – If true, only the anchor part is returned
- **time** (*datetime or None*) – If not None, the timestamp is included within the URL

class hf.module.**ModuleProxy**(*ModuleClass*, *instance_name*, *config*)
The access class to actual instances of the module class.

Since the module instances have to be independant and thread-safe for rendering, the common state is stored in the proxy, which creates a specific, run-time dependant module object for rendering.

acquire(*run*)

getModule(*run*)
Generate a module instance object for a specific HappyFace run.

isAccessRestricted()

isUnauthorized()

prepareAcquisition(*run*)

hf.module.**getColumnFileReference**(*table*)
Get a list of columns for a table that point to a file in the archive directory. :param table: Table to get the file columns for :ptype table: string or Table :returns: list of column names

hf.module.**moduleClassLoaded**(*mod_class*)

hf.module.**importModuleClasses**()

hf.module.**getModuleClass**(*mod_name*)
Get the module class for a given module name. :param mod_name: Name of the module :ptype mod_name: string

class hf.module.ModuleBase.**ModuleMeta**(*name*, *bases*, *dct*)
Meta Class of ModuleBase.

Its purpose is checking for the declarative module specification as described in *Special Class Variables*, as well as registering module classes in the system and creating additional variables based on the declaration.

hf.module.database.**module_instance**
Test

hf.module.database.**hf_runs**
123

6.8 hf.plotgenerator – Custom Plots for the Web Output

hf.plotgenerator.**init**()
Configure matplotlib backends by hf-configuration. Call before any plot-commands

hf.plotgenerator.timeseries.**getTimeseriesPlotConfig**(***kwargs*)
Extract the plot configuration from the URL arguments.

Possible arguments are a subset from the ones of timeseriesPlot(), see there for details.

Parameters

- **curve_XXX** – colon-separated curve info: (module_instance,[subtable],expr,title)
- **legend** – Show legend in image
- **title** – (string) Display a title above plot
- **ylabel** – self-explanatory
- **start_date** – Start date (Y-m-d)
- **end_date** – End date (Y-m-d)
- **start_time** – Start time (H:M)
- **end_date** – End time (H:M)
- **renormalize** – (true, false / 1, 0) Scales all curves to a [0,1] interval

Returns

A dictionary with the configuration. The entries are

curve_dict A dictionary encoding the curves to plot. Only curves are given where the user is authorized. to access. The key is the curve name, the value is a tuple of the following format: *(title, table, module_instance, col_expr, subtable)*

title Title of the curve. Same as given in the URL.

table sqlalchemy Table instance.

module_instance Name of the module instance to plot data from. Same as given in the URL.

col_expr The expression to plot. See *Column expression layout* for syntax.

subtable Subtable to plot from, empty string if plot from module table is requested. Same as given in the URL.

title Title of the plot

legend True or false, 0 or 1, flag to indicate if legend should be displayed.

ylabel Label of the y-axis

timerange Tuple of datetime objects, *(start, end)*

renormalize If *True*, each curve is scaled into a [0,1] interval.

auth_required Flag if authorization is required for any curve.

errors A list of error messages that occured during the function call

`hf.plotgenerator.timeseries.`**`timeseriesPlot`**(*category_list*, ***kwargs*)

Supported arguments (via **kwargs):

Parameters

- **curve_XXX** – colon-separated curve info: (module_instance,[subtable],expr,title)
- **filter** – include only rows where specified column in result set matches value, can be specified more than once: col,value
- **filter_XXX** – include only rows where specified column in result set matches value for curve XXX, can be specified more than once: col,value
- **exclude** – include only rows where specified column in result set matches value, can be specified more than once: col,value
- **exclude_XXX** – include only rows where specified column in result set matches value for curve XXX, can be specified more than once: col,value
- **legend** – Show legend in image
- **title** – (string) Display a title above plot
- **ylabel** – self-explanatory
- **start_date** – Start date (Y-m-d)
- **end_date** – End date (Y-m-d)
- **start_time** – Start time (H:M)
- **end_date** – End time (H:M)
- **renormalize** – (true, false / 1, 0) Scales all curves to a [0,1] interval

6.9 `hf.url` – URL utilities

`hf.url.`**`absoluteUrl`**(*arg*)

Decorator! Take an URL that is relative to the root URL of happyface and make it absolute respective to that root URL

`hf.url.`**`get`**(***kwargs*)

Generate a GET line from a dictionary

`hf.url.`**`join`**(*url*, *suffix*)

hf.url.**staticUrl**(*file*)

6.10 hf.utility – Miscellaneous Utility Functions

hf.utility.**addAutoLinks**(*string*)
Searches for URLs in string using a regular expression and adds <a>-Tags around them.

hf.utility.**matheval**(*input*, *variables={}*)
A safe way to evaluate mathematical expressions.

The following Python functions are available

- abs
- sin
- cos
- sqrt
- pow
- floor
- ceil
- max
- min
- float
- int

Parameters

- **input** – String with mathematical expression
- **variables** – A dictionary with variables available for input expression

Returns The result of the expression, or *None* in case of error.

hf.utility.**prettyDataSize**(*size_in_bytes*)
Takes a data size in bytes and formats a pretty string.

CHAPTER

SEVEN

TOOLS

7.1 Standard Tools

7.1.1 dbupdate

7.1.2 modconfig

```
-h
Display help message

-no-comment
Do not include comments in generated configuration
```

7.1.3 shell

7.1.4 gensiteconfig

7.2 Tool API

CHAPTER

EIGHT

DOCUMENTATION

The documentation you are currently reading is generated using Sphinx, the standard Python documentation generator that is also used for the Python reference manual.

It works alot like documentation generators for other languages, like Doxygen for C++ or JavaDoc for Java, using a bunch of text files in the *docs/* directory, as well as formatted docstrings inside the HappyFace source code as a basis to generate documentation in different formats. By default, Sphinx can generate documentation as plain HTML and LaTeX, as well as more exoctic formats like HTML-help, qthelp and JSON objects. The markup-language used by Sphinx is an extended verison of reStructuredText (reST).

Note: This is **not** intended to be a Sphinx Tutorial! Only the common commands and guidelines to write and generate HappyFace documentation are presented. You are urged to read more about

---

8.1 Building HTML

This is probably the most common option. Open your favourite shell and cd into the *docs/* directory. To generate the documentation, issue the following command

```
make html
```

That's it! You will now find the documentation in *docs/_build/html*.

Note: Because HappyFace is written in Python 2, you have to use the Python 2 version of Sphinx. Luckily, at the time of writing, the default Python version in most Linux distribution is Python 2.7, so you don't have to worry.

If you use for example ArchLinux or a Gentoo with Python 3, you have to set the environment variable SPHINXBUILD to sphinx-build2 and call *make* with the *-e* flag.

8.2 Building LaTeX

All information from the previous section applies here! With that in mind, just issue

```
make latex
```

This will generate a bunch of files in *docs/_build/latex*, including a Makefile. To generate either a PostScript or PDF version of the documentation, call one of the following makes. Of course, you can also generate both, if you like.

```
cd _build/latex
make all-pdf
make all-ps
```

8.3 Writing Documentation

The basic procedure to write documentation is adding or extending **.rst* files in the *docs/* directory and include them from *index.rst* or some other file.

Here are some tips

- Make use of cross references! They make it easier to understand the overall structure.
- Look at other documentation files from HappyFace!
- Look at Sphinx documentation from other projects! There is usually a link to show the code that generated the current page on each page.
- Make sure the Python code is correct! Sometimes you wonder why the generated docs are wrong, often the Python code contains some error. Check the make output carefuly.

8.3.1 API Documentation

Generating documentation from the source code is very convenient, since you will have an always up-to-date documentation for source code without worrying about reflecting changes in the source to the docs.

The first is adding docstrings to classes and functions in the source code. Actually, this is optional, because Sphinx can also generate basic documentation for undocumented members, but this is of course of limited usefullness. These docstrings have to be formated in Sphinx compatible reST. You then have to add a *.rst* file in the *docs/* directory. For core documentation, the naming scheme is *core_MODULE.rst*. Don't forget to mention the file in *core.rst* or *index.rst* as applicable, otherwise it is not included in the generated documentation!

The content of your new module reST file is fairly simple, since we to use Autodoc Extension from Sphinx. For example, your file looks like

```
==================================
mod:`hf.example` -- Example Module
==================================

.. automodule:: hf.example
    :members:
    :undoc-members:
```

CHAPTER

NINE

LICENSE

HappyFace is licensed under the following conditions:

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   You must give any other recipients of the Work or Derivative Works a copy of this License; and

   You must cause any modified files to carry prominent notices stating that You changed the files; and

   You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

CHAPTER

TEN

DOCUMENTATION TODOS

Todo

what is it?

(The *original entry* is located in /usr/users/happyface/docs/docs/configuration.rst, line 110.)

Todo

Include information about CSS and JavaScript.

(The *original entry* is located in /usr/users/happyface/docs/docs/module_dev.rst, line 196.)

Todo

write step-by-step module development guide

(The *original entry* is located in /usr/users/happyface/docs/docs/module_dev.rst, line 221.)

CHAPTER

ELEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

h

INDEX

Symbols

A

C

D

E

F

G

H

I

J

M

P

R

S

T

U

W