Grid Configuration Monitoring (GCM) Web Portal Documentation

Thomas Low*

March 10, 2009

Abstract

This paper contains a documentation about the *Grid Configuration Monitoring Web Portal* developed at CERN for the *Worldwide LHC Computer Grid* (WLCG). It describes all components in detail, so that it should be easy to maintain, enhance or extend the portal.

Contents

*Grid Deployment, IT Department, CERN, Switzerland. thomas.low@cern.ch

1 Introduction

The aim of this work is to collect information about the configuration of the WLCG production grid infrastructure. It does this by providing a program that runs with a grid job that gathers information about a worker node and its environment. See the documentation of the *Worker Node Client*. [4]

A server [3] retrieves these information, so called *Tests* and stores them in an oracle database using *Django*. Finally this web application visualizes the data for statistical purposes or to help to resolve problems. Figure 1 illustrates the whole layout.

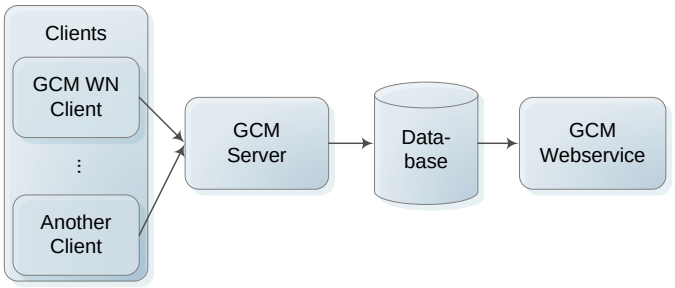

Figure 1: Layout of *Grid Configuration Monitoring* programs

The *Web Portal* is provided as a package which resides in the directory [`web/`] of the package `grid-gcm`.

2 The Web Portal

The *Web Portal* is written in *Python* and uses *Django* and its database backend. Thus it is advised to read *Djangos* tutorials [1] and the database documentation [3] before going on.

2.1 Configuration

As usual *Django* will be configured by a `settings.py` located at [`web/etc/settings.py`]. It contains only settings which affect the *Web Portal*. Database settings will be imported from [`db/etc/settings.py`] as described in the server and database documentation [3].

It is also possible to create a `settings_local.py`, so it is not necessary to customize `settings.py` which is under version control.

The script `grid-cm-web` acts exactly like *Djangos* `django-admin.py` except that you do not need to specify `DJANGO_SETTINGS_MODULE`.

So in order to start the development server use the following command:

```
grid-cm-web runserver 0.0.0.0:8163
```

2.2 Loading of Views

Once the *Web Portal* will be started all models will be loaded into the namespace `gcm.tests.models` as described in section 3.2 in the database documentation [3].

Then a special script [`/web/src/gcm_site/loader.py`] iterates over all those *Test* models and assigns to each of their attributes all generic views. In this case assigning means it registeres a URL in the `urlpatterns.py` which forwards a certain user request to a generic view. The URL pattern will be constructed by the name of the model and its attribute.

So basically everything works like usual in *Django* except that most URL patterns will be constructed dynamically from the database models.

To link all of them together there is a special index view in [`web/src/gcm_site/base.py`] which acts like a sitemap.

Example

Consider the database contains a *Test* model like this:

```
class WN_System_CPU(WN_TestModel):
    ProcessorCount = models.IntegerField("Processors",null=1)
    ProcessorCores = models.IntegerField("Cores",null=1)
    ProcessorClockSpeed = models.FloatField("Speed",null=1)
    ProcessorVendor = models.CharField("Vendor",max_length=100,null=1)
```

As soon as the *Web Portal* will be started it will find this model and iterate over its attributes. It then will link some URL pattern to generic views which are responsible for constructing the actual web page. Such a URL pattern might look like so:

- `wn_system_cpu/processorcount/` → `globalTestView`
- `wn_system_cpu/processorcount/region/<region_id>/` → `globalTestView`
- `wn_system_cpu/processorcount/site/<site_id>/` → `siteTestView`
- `wn_system_cpu/processorcores/` → `globalTestView`
- ...

When a user requests a URL, e.g. `wn_system_cpu/processorcount/region/5/`, *Django* will call the corresponding generic view with some arguemts like the users request and the id of the region: in this case a method called `globalTestView` with the argument 5. (amongst others)

2.3 Generic Views

There are 4 generic views located in [`web/src/gcm_site/views/genericFieldViews.py`] and [`web/src/gcm_site/views/genericTestView.py`]:

- **globalTestView** displays different groupings of *Worker Nodes*, e.g. lists of Regions, Countries, Sites.
- **siteTestView** displays a list of *Worker Nodes* in a certain *Site*.
- **wnTestView** displays all the tests which were produced by a certain *Worker Node* in chronological order.
- **testView** displays a certain test result.

2.4 Generic Graphs

There are 3 generic graphs [`web/src/gcm_site/views/genericGraphs.py`] which visualize data using Google Charts [2]:

- **getPie** displays a distribution in a pie chart. Only used by string fields.
- **getHistogram** displays a distribution in a histogram. Only used by number fields.
- **getAggregatorsOverTime** displays a graph wich visualizes how things changed over time. Used by both string and number fields.

2.5 Templates

The templates, which will be used to render html pages, are located in [`web/src/gcm_site/templates`]. They are arranged in a hierarchie which has its root at [`web/src/gcm_site/templates/base.html`]. Each template then only defines the differences to a template it is derived from.

Templatetags

There are only two small additional templatetags defined in [`web/src/gcm_site/templatetags`]:

- **in** is used to check wheter a value is in a list.
- **code** is used to highlight source code using the python module pygments [5].

2.6 Virtual Test Models

Virtual *Test* models are used to preprocess results from the database before visualizing them on the *Web Portal*.

Example

Consider having a *Test* which gets information about the operating system.

```
OperatingSystemName:ScientificCERNSLC
OperatingSystemRelease:4.5
OperatingSystemVersion:Beryllium
```

But there are multiple ways to visualize this data. For example it might be useful to see the distribution of operating systems ignoring or respecting their release numbers. On the other hand it would be redundant to store this data like so in the database:

```
OperatingSystemName:ScientificCERNSLC
OperatingSystemNameRelease:ScientificCERNSLC 4.5
```

With virtual *Test* models and oracle database views it is possible to use both variants in the *Web Portal*. Therefore create additionally to the most effective database model another one and mark it as *virtual*:

```
class WN_System_OS(WN_TestModel):
  OperatingSystemName = models.CharField(max_length=100,null=1)
  OperatingSystemRelease = models.CharField(max_length=100,null=1)
  OperatingSystemVersion = models.CharField(max_length=100,null=1)

class WN_System_OS_Virtual(WN_TestModel):
  OperatingSystemNameRelease = models.CharField(max_length=100, null=1)

  class TestMeta(WN_System_OS.TestMeta):
    virtual = WN_System_OS
```

Note that you have to create those database views by yourself. *Django* does not support them which also results in problems when using `syncdb`. You could create a database view in your favorite oracle client like so:

```
CREATE OR REPLACE VIEW "TESTS_WN_SYSTEM_OS_VIRTUAL" AS
  SELECT *,
    TRIM("OPERATINGSYSTEMNAME" || ' ' || TO_CHAR("OPERATINGSYSTEMRELEASE"))
      as "OPERATINGSYSTEMNAMERELEASE"
  FROM TESTS_WN_SYSTEM_OS;
```

3 References

[1] Django. Djangos tutorials. http://docs.djangoproject.com/en/dev/intro/.

[2] Google. Charts api. http://code.google.com/apis/chart/.

[3] T. Low. Grid configuration monitoring - server and database documentation. See https://twiki.cern.ch/twiki/bin/view/EGEE/WorkerNodeConfiguration#Documentation.

[4] T. Low. Grid configuration monitoring - worker node client documentation. See https://twiki.cern.ch/twiki/bin/view/EGEE/WorkerNodeConfiguration#Documentation.

[5] Pygments. Pygments. http://pygments.org/.