LHCb Performance and Regression

Internal Note

Issue: 1
Revision: 0

Reference: unknown
Created: February 6, 2013
Last modified: May 26, 2013

Prepared by: Emmanouil Kiagias

Abstract

Put your abstract here.

Document Status Sheet

1. Document Title: LHCb Performance and Regression			
2. Document Reference Number: unknown			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
Draft	1	February 6, 2013	First version

Contents

List of Figures

List of Tables

1 Introduction

What is **LHCbPR**

LHCb Performance and Regression(PR) is a service designed to record important measurements about releases of the LHCb application. Applications , such as Gauss, Brunel, Moore etc, receive input in the form of configuration files and produce, as an output, various information. LHCbPR is **not** intended to actually run the jobs (maybe in the future as an extra feature), but instead to manage and track the bulk of information produced by them.

The LHCb Performance and Regression is a framework that allows LHCb software developer to push information for a run of their code(job charactetistics, results, performance measures, files) to a central database, from which an analysis can be performed across version, configs(platforms) etc.

Why LHCbPR

In the past, in order for a user working group to run a job, they had to execute a job for a specific application -using a configuration file-, gather the results and then run their own analysis on the

produced sample. Finally, when such a process was complete, they needed to find a way to publish their results. Often ending up with serving static documents such as HTML, CSV or e-mail.

LHCbPR was conceived as a tool to reliably organize the process of configuring and monitoring a job execution. The framework, solves the problem of gathering the results of a job execution, by providing the user a complete script that, when executed, can produce the desired output, according to a configuration file. This script, also includes a list of **handlers** that collect the defined aspects of the job results and push them to the database in a uniformed way.

By collecting the results in such an organized manner, it provides the solution to the second problem, mentioned above; the running of the analysis. Since, all the data are stored as objects in the database, the framework provides an abstract and easy way to deploy algorithms and functions which perform analysis on the saved data.

Finally, LHCbPR handles the presentation of the collected data by providing a set of templates for the creation of web pages specific to each analysis and customized to the preferences of each user.

Users of LHCbPR

The LHCbPR users can be divided into three categories: Administrators, Application developers and End users.

The administrators group is responsible for the integrity of the collected data and the efficiency of the service. They maintain and support the system, making sure the application is functional and available to the end users.

Application developers are the users who actively design and develop modules for the framework, thus extending the functionality of the application.

Finally, the end users are the main body of the users of the service. They put the tools provided by the application developers to practical use and their job results are populating the database.

2 Architecture

2.1 Django

LHCbPR framework is written in python and the main body has been has been deployed with **Django** . Django is an **open source** web application framework, written also in Python. The application is hosted in a shared Apache web server with fcgi module. For more information about apache, fcgi and django can be found in their corresponding web pages.

2.2 3TP - 3-Tiered Programming

The LHCbPR follows the architectural model of three tiered programming. **3TP** provides a way to theoretically categorize the components of an application, providing abstract modularity. In essence the 3TP model is a client-server architecture which is composed of three layers. Namely, the presentation tier, the functional process logic tier and the data storage tier. These tiers communicate with one another in a strictly defined way, which allows the developer to independently maintain each tier. In most practical applications of the model the presentation layer includes all forms of user interaction with the service. The process logic tier encompasses the whole of the application functionality, while the data storage tier handles the flow of the information from and to the data source (in most cases a database).

The theoritical model mentioned above is practically implemented in LHCbPR(see Figure 1) and is explained in detail in the following sections.

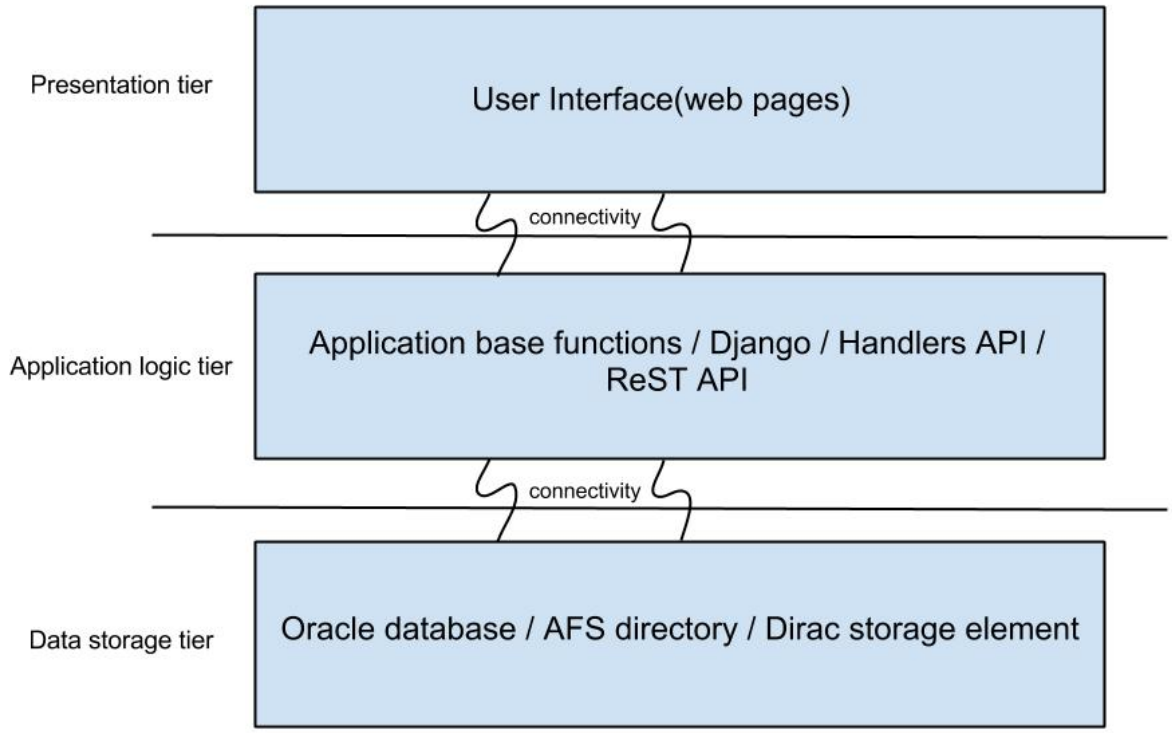

Figure 1 3TP implemented in LHCbPR

2.2.1 Presentation layer in LHCbPR

The presentation tier of LHCbPR is composed of a set of web pages that utilize a **ReST** API, and the handlers' interface.

ReST comes from Representational State Transfer and provides an abstract way of stateless client-server communication. Django's capability of handling URLs, through the stateless HTTP Protocol, serves as a powerful tool to design and develop ReST Interfaces. LHCbPR's API functions range from generating and managing unique identifiers(UID) for objects in database, to compiling scripts to run a simulation and collect their output.

Above the ReST API exists a set of web pages which are served to the end user by the django framework. These mainly consist of static HTML styling code and javascript/jquery functions that allow asynchronous communication with the ReST Interface. This web application provides an easy and visual way for the handling of the information stored in the database.

The handler's interface is different to the ones mentioned above, in the sense that it provides a programmatical way to communicate with the service. That is, an application developer can use the handler library functions to strictly define the amount and type of data that will be collected from a job execution. Essentially the handler is an independent utility that can be called after any job execution to parse, group and collect specific information from the output.

2.2.2 Application logic layer in LHCbPR

The application logic tier a set of python modules that can be divided into two categories that each serve its own purpose; the base application logic and the database connectivity.

The base application logic is a group of functions which handles the incoming users' requests, sanitizes their input and generally tries to accomplish the application's objective. Also it provides an interface for application developers to specify the way they process the data that is stored in the database, by developing python modules for analysis. The development of such modules is abstracted from the specific implementation of the framework which makes the analysis process much simpler and more straightforward.

The database connectivity is achieved by utilizing the models/views-function of the django framework. The representation of the content of the database as logical objects allows easy access and management of the stored data. Also it acts as a powerful mechanism for enforcing the db schema.

2.2.3 Data storage layer in LHCbPR

The third and last layer of LHCbPR is responsible for the data management of the whole service. It consists of three independent data sources.

The first and most important is an Oracle database which is provided by the IT central Oracle service. It contains every bit of information about the jobs that have been executed, including the statistics and output information, apart from files. In case where whole files, produced from a job(e.g ROOT files), are needed to be saved then they are not handled from the database but from a filesystem.In the database is only saved a relative path, to a media directory, for each file .

The files mentioned are stored in an AFS shared directory for easy access and robust backup.

Finally the last data source is another shared directory which is accessed through a DIRAC storage element and is used as a temporary caching space between the server and the data collected by the handlers.

3 Database

In order to describe a job execution through the LHCbPR framework and generate a script which will run the defined-described job, **job descriptions** are saved/handled as objects in the database. Also they are used to identify which data comes from which job execution. The database objects will be explained in details in the following subsections.

Here is a summary of the database schema tables:

Jobattribute
JobDescription
JobHandler
JobResults
Options
Platform
Requested_platform
ResultsString
ResultInt
ResultFloat
ResultFile
SetupProject

3.1 Job description object

The tables which constract a job description are the following:

- JobDescription, Application, Options, SetupProject

The Application table contains two columns : appname(application) and appversion(application version). This pair is unique for each application and ,as the name says, this table represents the application(eg Gauss, Brunel etc) with a specific version(eg v42r0, v43r2p0 etc) and its data will be used to setup the environment to prepare a job execution for the corresponding application.

The Options table represents the options which we will used to execute a job and it has two columns: description and content. The description column is more or less a name(alias) to identify the options and the content is the actual content of the options, which can be a path to an options file or a big string containing the actual raw options. Note that the pair description-content in the table must be unique, can not have the same description for two(or more) different contents or the same content for two(or more) different descriptions.

The SetupProject table represents the extra arguments(optional) to the SetupProject command. Also this table has two columns : description and content. The description is a name(alias) to identify the SetupProject extra arguments(eg –no-user-area) and the content is the actual extra arguments. Note the pair description-content must me unique(same as the pair description-content of the Options table)

All the three previous tables constract a job description object. This is the information which is needed to properly configure and execute a job, this is the table (see Figure 2).

3.2 Job

Once there are saved job descriptions in the database they can be used to actually run jobs. The Job table has the following columns: The jobDescription_id, an id to specify which job description object was used to execute the job(application-version, options etc), the time when the job started and finished(time_start and time_end columns), the status of the job(finished, running etc), success(whether the job was successful or not). Also a job contains the platform(cmtconfig) and the host(machine) on which it was executed. Apart from the others, which are single columns, the platform and the host columns(and the jobDescription) are ids mapping to the corresponding tables Platform and Host.

The Platform table contains only one column containing a platform(cmtconfig).

The Host table represents a machine(on which jobs are executed) and has the following columns: hostname(name of the machine eg pclhcb10), cpu_info and memory_info for the machine.

All the above constract a job object which represents a single job execution(see Figure 3)

3.3 Job results

Up to now we described how a job description and a job execution are represented in the database. Now we will see how job results are saved. Each job execution produces various results such as log files, ROOT files(histograms), xml files etc. A result can be a number(a float, an integer), a string or a whole file. Each result is considered to be a Job Attribute and it is saved in the JobAttribute table.

The JobAttribute table represents,identifies a job result and has the following columns: The name column which is the name the result will hold(eg totalCrossSection), the description(optional) column which is a description for the attribute, group column(optional) which specifies a group for the attribute(eg group: ”Timing”) and the type column which can be an Integer,String,Float or File.

The ResultInt,ResultString,ResultFloat,ResultFile tables ”inherit” the JobAttribute table and just provide the data column for each job attribute. So the JobAttribute table works as the parent table and

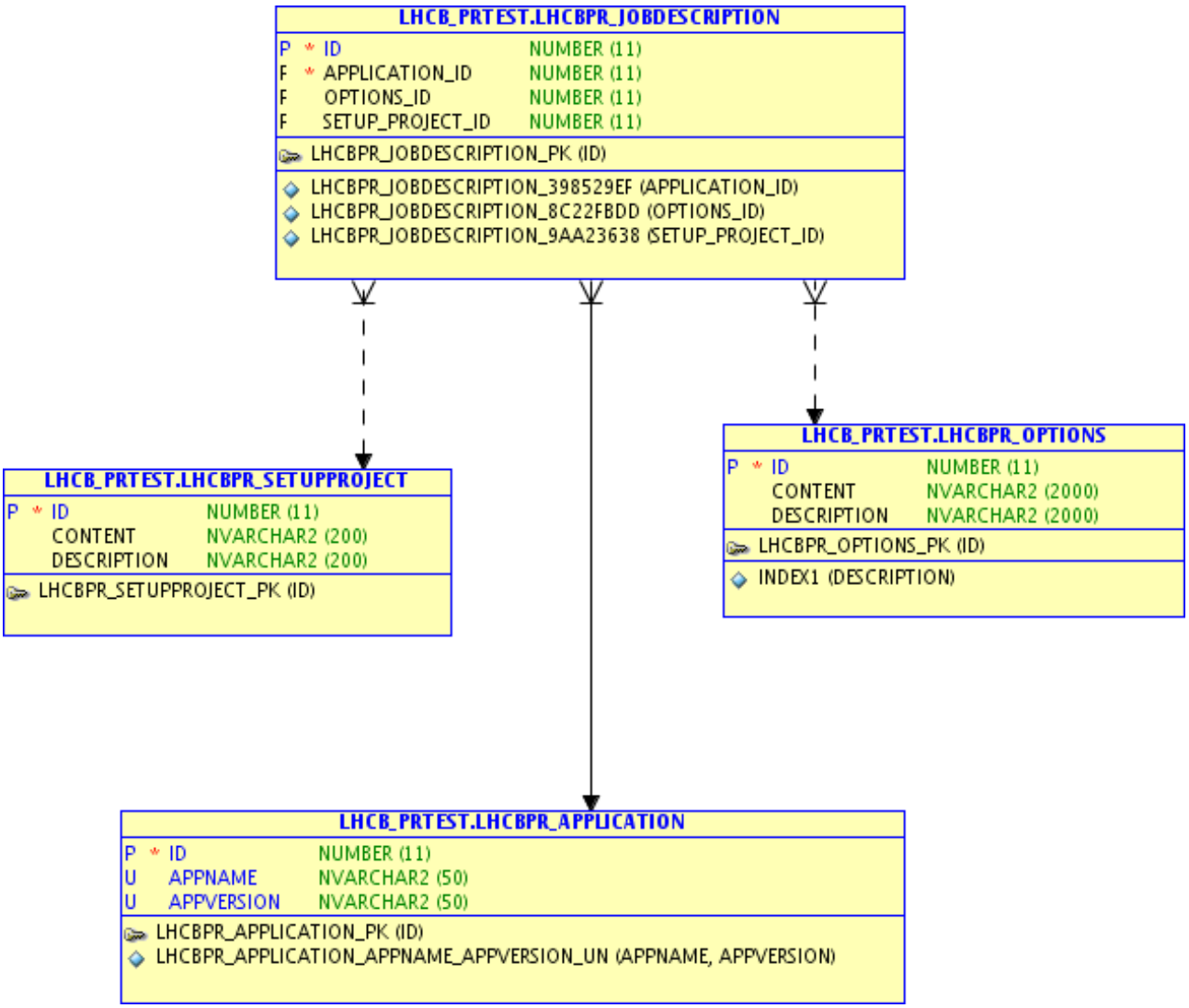

Figure 2 Job description table summary

depending on the type column it has a corresponding child table, example: if the type of a job attribute is ”Integer” then the it will have as its child a ResultInt.

To associate the previous tables, one with each other, and all together with the job table a middle JobResults table is used which contains a JobAttribute_id mapping to a JobAttribute object and each Result(ResultString etc) table contains an id mapping to a JobResults object(see Figure 4). Finally the JobResults table also holds a job_id which maps to a specific job.

So for each job execution there are multiple JobResults instances containing the collected results.

3.4 Handlers

Handlers are python modules which take as a parameter a directory and collect results out of it. Handlers can collect many kind of data such as Numbers(Floats, Interger) ,Strings, even save a whole file. In order to associate handlers with jobDescriptions and job executions the following tables are used: Handler, JobHandler, HandlerResult

The Handler table represents a handler python module and contains two columns: a name column which is the name of the python module and a description column which is a description(optional)

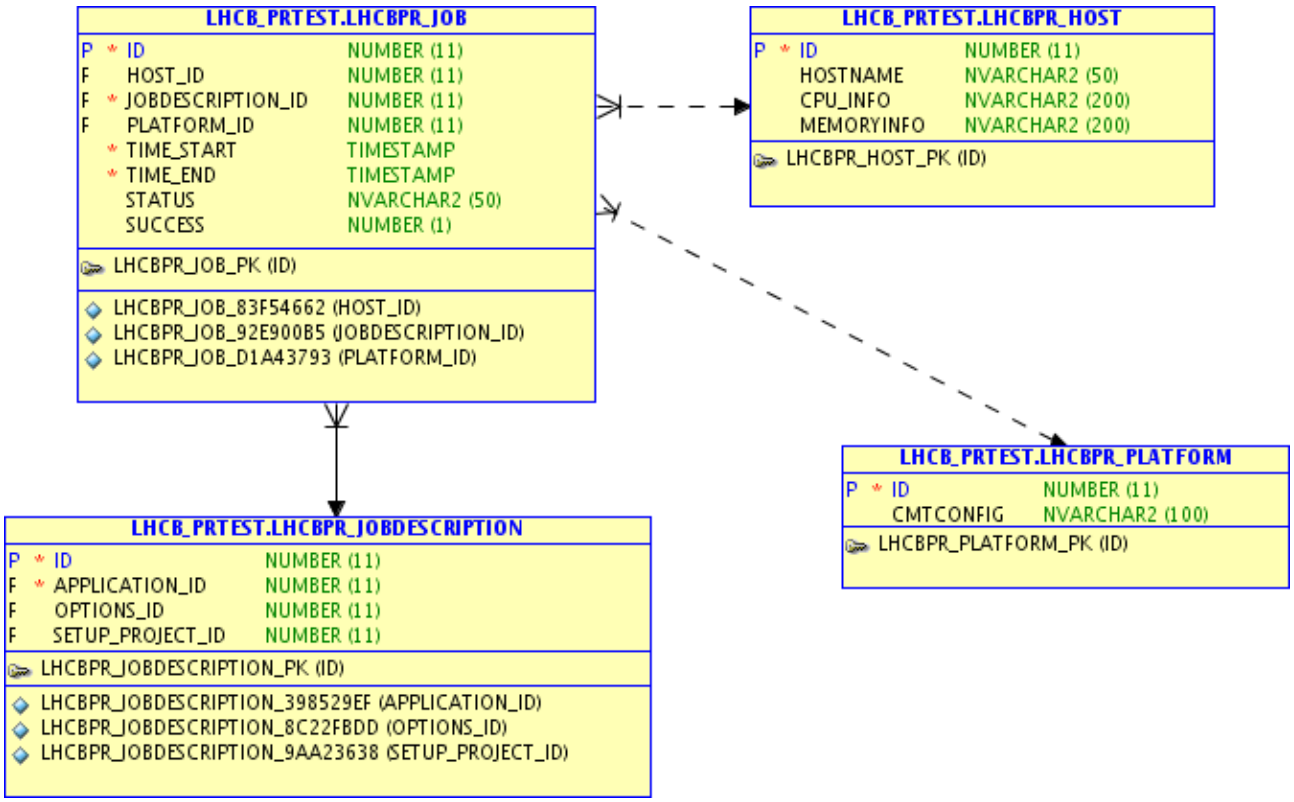

Figure 3 Job table summary

about the handler. A handler can collect data from different jobDescriptions even from jobDescriptions with different applications (example the TimingHandler works for all applications). A handler is associated with a jobDescription through the JobHandler table.

The JobHandler table is used as a middle table(many to many relation) between the JobDescription and the Handler tables.

So for each JobDescription there are associated handlers which will collect data after a job execution. When a job is finished, the handlers(may be just one) collect data and produce a zip file with the results. In some cases though a handler can fail(either because of a user's mistake or because a job execution did not produce the results it should have produced), so we keep track of the handlers by using the HandlerResult table.

The HandlerResult table shows whether a handler was successful or not for a specific job execution, so it contains a handlers_id(to map a handler object), a job_id which maps to a job execution and finally a column success to specify whether the handler was successful or not for this job.

For the above tables see Figure 5.

4 Implemented functionality

Here follows a summary of the implemented functionality of the LHCbPR.

4.1 Job Description Management

- Create new Job Description

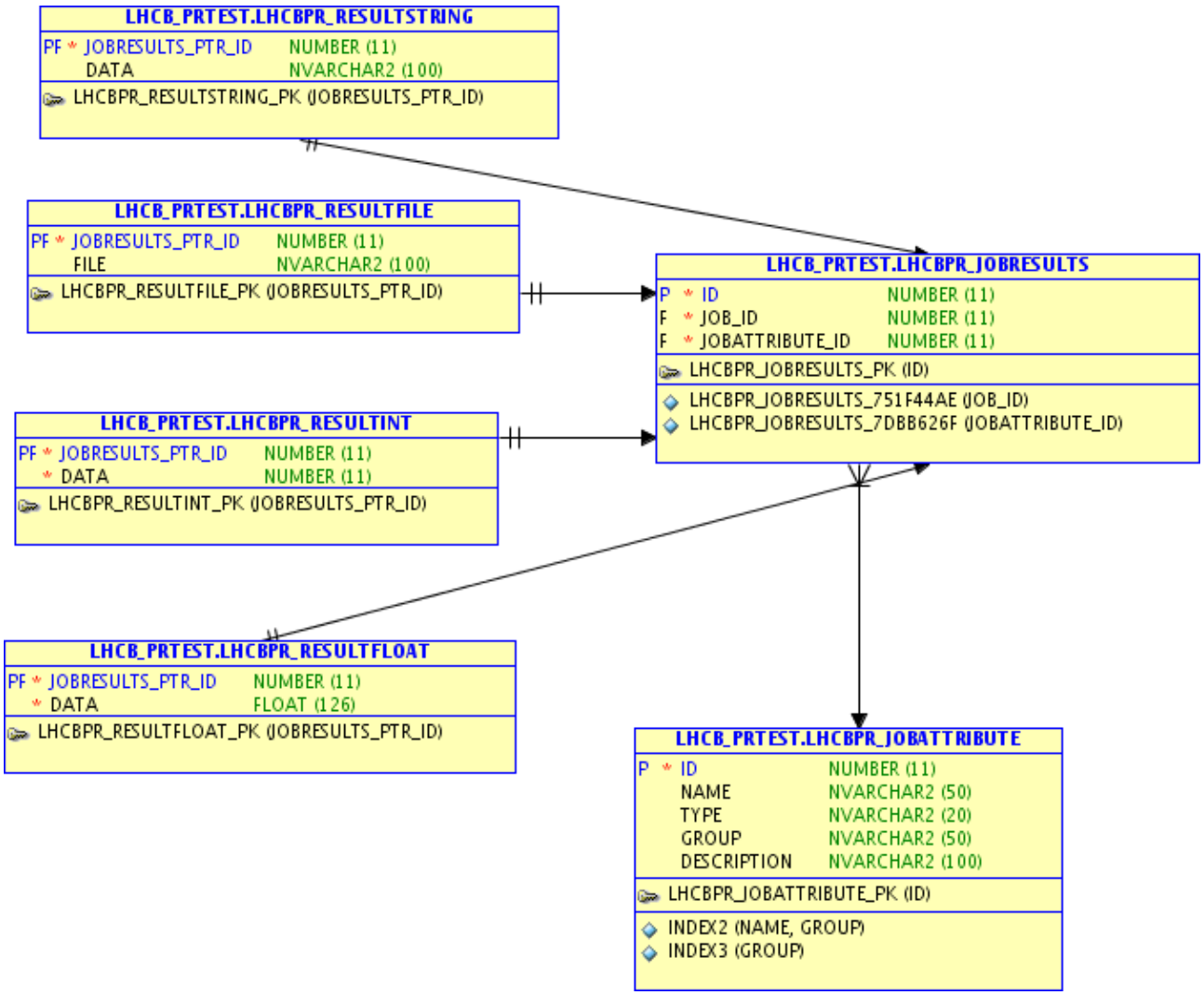

Figure 4 Job attribute table summary

A new job description can be created through a web interface by cloning an existing job description(cloning means to create a new object based on a preselected job description).

- Update Job Description

  A job description can be updated(change one or more attributes) through a web interface. Warning: a job description can not be edited if there is at least one executed job saved in the database with this job description.

- Create Job Description from script

  A job description can also be created from a script(by making an HTTP request with the desired attributes). In case the job description exists the function returns the existing job description id.

- Generate Job script

  A script which will execute a job using a selected job description. The script can either be created through a web interface(by clicking a button) or by making an HTTP request through a script.

4.2 Job Handling

- Basic python handler interfaces

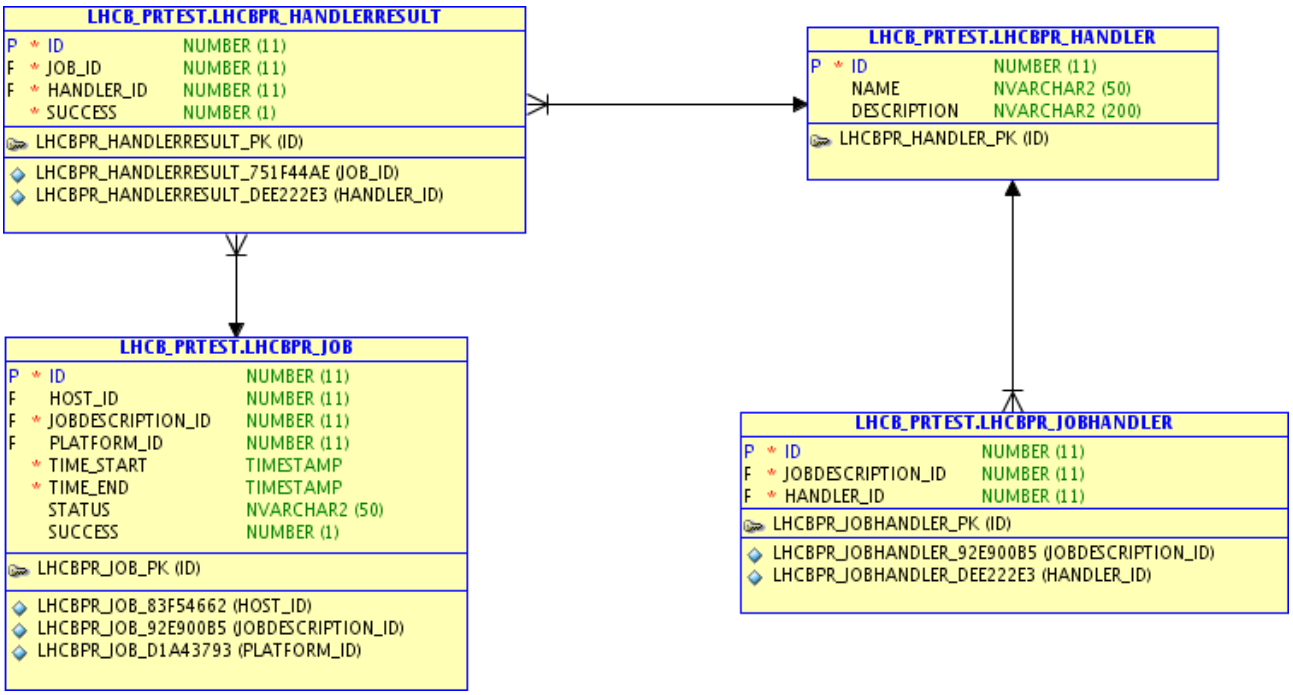

Figure 5 Handler table summary

An application developer can use the provided handler functions to strictly define the amount and type of data that will be collected from a job execution.

4.3 Analysis

Here follows a summary of the already implemented analysis pages

- Basic analysis

  In this analysis page the user can select a single quantity(a single attribute from the database eg EVENT_LOOP) also select a job description(or part of it eg only specific options or platform) and calculate the average value, standard deviation, generate a histogram for the corresponding selections. Up to 3 histograms can be generated at once and also be superimposed.

- Timing analysis

  In this analysis the user can select a job description and generate a tree containg the average timing performance of the algorithms of all the available job runs for the selected job description.

- Histograms analysis (available only for Gauss application)

  More or less the same as the basic analysis but in this case the user selects a histogram title(from the saved ROOT files) instead of a single quantity, and calculate the average histograms for all the available ROOT files. Also the histograms can be superimposed or divided(for the division there must be exactly 2 histograms , no more no less)

- Trend analysis (not fully completed yet)

  In this analysis the user can select a single quantity and perform an analysis across the versions of the application.

4.3.1 Custom Analysis Framework

Each analysis page is a module and all the above analysis pages are deployed using a small custom analysis framework. The development of such modules is abstracted from the specific implemen-

tation of the LHCbPR framework which makes the analysis development much simpler and more straightforward for the developers.

4.4 Authentication

All the LHCbPR's pages and services are protected by shibboleth, the user must provide a valid cern account in order to access the service. Also there is the ability to restrict the access to only specific groups ,if necessary, in a desired service/services.

There is also restricted access to the administrator panel. The admin panel can be accessed only from superusers which are defined inside the database through Django. For more info about superusers check the Django documentation

4.5 Administration

The LHCbPR also comes with a default administration panel provided from the django framework to manage database objects through a web interface. Also the LHCbPR provides a function to either delete a job by id or hide a job(flag to hide bad data, these jobs will not be used in analysis pages). These functions can olny be accessed from the administrators. As mentioned in the above subsection only superusers can access the admin panel. For superusers check Django official documentation

5 Analysis development

The LHCbPR framework provides also the ability to develop new analysis modules and handlers. In this section will be descrided how to develop a new analysis module (a functional analysis web page) using a small custom framework

5.1 Getting started

Below are described the steps you need to do in order to develop a new analysis module locally on your computer and test it.

First get the latest version of the project from the LHCb git repository:

git clone /afs/cern.ch/lhcb/software/GIT/LHCbPR.git

Then run the script :

LHCbPR/devel/getdjango (version of django is set, hardcoded, inside the script)

This will download the Django source and place inside the LHCbPR/devel directory . The LHCbPR is deployed with Django framework so you need a copy of it in order to run the service locally.

After that you need tou setup the enviroment in order to use the cx_Oracle module from afs:

Note: the script works on **bash** not in tcsh/csh etc.

source LHCbPR/devel/setuptools

The LHCbPR uses an oracle database so it needs the cx_Oracle to establish a connection with it(in case you have not executed the getdjango script the setuptools script will invoke it to download the django for you).

Attention. One more step to finish the setup. You must provide a **myconf.py** file in **LHCbPR/django_apps/** directory in this format:

```
dbname = 'name_of_the_database'
dbuser = 'username_for_the_database'
dbpass = 'database_password'

key = 'key_django_needs'

#ATTENTION
# LOCAL must true in order to run the project locally
LOCAL = True

#also debug true is needed to get the django's debug page
#in case an error happens
DEBUG = True

#root url for the local development needs to be /
ROOT_URL = '/'
```

Note: Ask the administrator to give you the needed information (dbname,dbpass etc). If you have access to $LHCBSOFT/webapps/LHCbPR/LHCbPR/ copy the **myconf.py** file from there (remember to use the testing or developing database not the production one when deploying locally). In case you copy the file from $LHCBSOFT do not forget to change the settings as shown above (eg LOCAL variable to LOCAL = True, ROOT_URL = '/' etc).

And once you have the myconf.py file **everything's ready** in order to start running your local copy (you must keep the runserver command running to make your local LHCbPR work):

LHCbPR/django_apps/manage.py runserver

Do not forget to run the **source LHCbPR/devel/setuptools** everytime you change **shell**(eg: work the project in a new terminal) in order to setup the enviroment with the needed tools(without this script the runserver command will crash).

5.2 Writing a new analysis module

Note: Before you continue it is suggested that you take a look about **django templates** (in case you are not familiar), more info can be found **https://docs.djangoproject.com/en/dev/topics/templates/**

5.2.1 Initialization

Once you have downloaded a local copy of LHCbPR project from git and set up the enviroment you are ready to start deploying your new analysis module. First create a **new folder** with the name of your new module under to **LHCbPR/django_apps/analysis** directory. Let's assume your new analysis module will be called **hello_analysis** , then create a folder :

mkdir LHCbPR/django_apps/analysis/hello_analysis

Then create an __init__.py under your new created directory(your analysis code will reside in there) :

vim LHCbPR/django_apps/analysis/hello_analysis/__init__.py

Each analysis module must have two basic **templates** in the templates directory:

- **render.html**
- **analyse.html**

The **render.html** is the one that will be used to diplay the contents of your analysis page and the **analyse.html** is the template that will be used to diplay the actual analysis results(this template is loaded inside the render.html automatically), create them in the templates' directory under a directory having the name of the analysis(same we did with the module) :

mkdir LHCbPR/django_apps/templates/hello_analysis

vim LHCbPR LHCbPR/django_apps/templates/hello_analysis/render.html (2)

vim LHCbPR LHCbPR/django_apps/templates/hello_analysis/analyse.html (this is not mandatory for the beggining)

The render.html template has to follow a skeleton structure. The skeleton is available at:

vim LHCbPR/django_apps/templates/analysis/skeleton_analysis.html (1)

Note, to copy the skeleton to you render.html use: **cp (1) → (2)**

The above skeleton can be extended or use as is. Instructions on how to use or extend the skeleton are in the file itself.

The **docstring** of the __init__.py file(your module's docstring) will automatically be used as **help-text**(help div in the template) in your new analysis page. In case you dont provide a docstring then the helptext will be "This module is not documented yet". In case you want to hardcode the helptext inside your render.html template then follow the instructions in the skeleton_analysis.html.

The **title** of your analysis page will automatically be the selected application followed by the name of your module, so in our case of hello_analysis module and GAUSS application, the title of the page will be "GAUSS hello_analysis analysis". In case you want to **override** the default generated title, just provide a variable by the name title in your __init__.py file:

```
#this will be now your page title
title = "This is my page title"
```

Edit the __init__.py file in your analysis directory and add the 3 following needed methods :

- **def isAvailableFor(app_name):**
- **def render(**kwargs):**
- **def analyse(**kwargs):**

5.2.2 isAvailableFor(app_name):

Once you have created your analysis module(folder etc), as mentioned above, it's time to specify for which applications the analysis is available for. For example the "histograms" analysis is available only for the GAUSS application so the __init__.py file of the histograms module has the following method:

```
def isAvailableFor(app_name):
    if app_name in ['GAUSS']:
        return True

    return False
```

Everytime you select an application from the analyse page, the LHCbPR "walks" in the analysis' module directory and calls in a sequence the isAvailableFor(app_name) method from every module. If the method returns "True" then the analysis will automatically appear as button in the rendered page, else if it's "False" it wont appear. So in case you want to make your analysis available for all applications you can just do:

```
def isAvailableFor(app_name):
    return True
```

In that way, no matter what the app_name variable contains the method will always return True so it will appear to all applications

*5.2.3 render(**kwargs)*

The **def render(**kwargs)** function returns the **render.html** to the web browser, which resides in the templates/hello_analysis/directory, along with a dictionary with all the variables you want to use in your render.html template, for example:

```
def render(**kwargs):
    myvariable = "test_string"

    return { 'myvarialbe' : myvariable }
```

This will return the render.html template along with the myvariable which you can access it in the template with: {{ myvariable }}. Before you continue please make sure you have checked how a render.html file must be in **LHCbPR/django apps/templates/analysis/skeleton_analysis.html**.

As it is mentioned in the skeleton_analysis.html your render.html inherits the Versions, Options, Platforms, Hosts selections menu. By default the LHCbPR returns the Versions, Options, Platforms, Hosts objects for the successful jobs of the corresponding application.

Note, for example in the **timing analysis** we need, not only the objects for the successful jobs but also we need the objects who also have jobAttributes with some specific groups, so in that case we override the default values which LHCbPR returns.

So if you want to return different objects then you can **override the default values** by providing 'options', 'versions', 'platforms', 'hosts' in your returned dictionary. Here follows an example of how we override the default values in the timing analysis:

```
#missing code here ...
#...
from lhcbPR.models import Host, Platform, Application, Options

def render(**kwargs):
    #some missing code here
    #....
    app_name = kwargs['app_name']

    #some missing code here, for full version,
    #you can check the LHCbPR/django_apps/analysis/timing/__init__.py
    #....
    options = Options.objects.filter(jobdescriptions__jobs__success=True,
        jobdescriptions__application__appName=app_name,
        jobdescriptions__jobs__jobresults__jobAttribute__group='TimingTree').disti

    versions = ...

    platforms = ...

    hosts = ...

    dataDict = {
```

```
                'platforms' : platforms,
                'hosts' : hosts,
                'options' : options,
                'versions' : versions,
               }

    return dataDict
```

In that example you can see that we override the options,platforms etc by providing our own objects. That way the template will use these objects instead of the default ones.

Important: if you are not familiar with the above overriding example and you wish to override the default values for your analysis then it is suggested you take a look in the **django models** at : **https://docs.djangoproject.com/en/dev/topics/db/models/**

In the **render(**kwargs)** you can access the following data:

```
# you can take the application from the url(such GAUSS, BRUNEL etc)
app_name = kwargs['app_name']

#the data which request contains,
# same as doing ,in case of POST request: requestData = request.POST
requestData = kwargs['requestData']

#and finally, in case you need to have access to the full request object
myrequest = kwargs['request']
```

*5.2.4 analyse(**kwargs)*

This function will automatically be called when you press the "Retreive results" button from your analysis page and here you need to implement what shown as results of the analysis. Same as the render function you can access the same data from the kwargs as shown exactly above(app_name, requestData and request). Once you click the "Retreive results" button the page will automatically make an **ajax request** to your analyse(**kwargs) sending the selected Versions, Options, Platforms, Hosts(comma separated ids) from the selection menu along with the data of your other custom elements Let's assume that you have these two elements in your render.html template:

```
<input id="firstName" type="text" value="Emmanouil">
<input id="lastName" type="text" value="Kiagias">

<input id="myradio" name="myradiobutton" type="radio" checked="checked">
```

The page will put the id and the value of your elements in the request dictionary along with the selected Versions, Options..etc object, so in this case the request dictionary will like this:

```
{
  'options' : 1,5,7, # comma separated primary keys of the selected options
  'versions' : 3,5 ,
  'platforms' : 45,7,9,
  'hosts' :78,
  'firstName' : 'Emmanouil',
  'lastName' : 'Kiagias',
  'myradio' : True
}
```

You can access these data from the kargs['requestData'] dictionary. Then in your analyse(**kwargs) function you can use these data to make queries to the database(check: from django.db import connection) and any other calculation you need to produce your analysis results and just return a dictionary with your data. The analyse function will load the analyse.html template along with your data in the render.html template. So you can visualize your results with javascript,jquery etc, or any other web tools in your analyse.html template and they will be loaded in your analysis when you press the "Retreive results button".

5.2.5 Retrieving results from database

Once you have the input data you need from your analysis page you can use it and query the database. Django provides various ways to query information from your database. For example way is through the **django models** (django model API, which in our case is not the preferred way unless your query is very simple, more info: **https://docs.djangoproject.com/en/dev/topics/db/queries/**) or another way is through the **python cursor**.

In most of our cases we will need to write an sql query and execute it through **the cursor**, then fetch the results from the cursor. Here follows an example of how the cursor works:

```
from django.db import connection
#
#...
#

def analyse(**kwargs):
    requestData = kwargs['requestData']
    #let's assume your and attribute id in your requestData
    atr_id = requestData['atr']

    #establish connection with db, get the cursor
    cursor = connection.cursor()

    #then execute your query(here is an example, not real functional)
    cursor.execute('SELECT * FROM lhcbpr_jobattribute WHERE id='+atr_id)

    #then get you resutls, either fetch them all
    #myresults = cursor.fetchall() #here it returns a list with the selected rows

    #or loop over one by one, preferred if you want to handle
    #each selected result independently

#get one result from the cursor
    result = cursor.fetchone()

myList = []

    #loop while there are no results left
    while not result == None:
        #do stuff here with the result
#...
#add my result in myList
myList.append(result)

        #read the next result
        result = cursor.fetchone()

    #once out of the loop, do stuff with my myList
```

```
    #...
    #then return my results to be displayed through my analyse template

    return { 'myresultsList' : myList }
```

IMPORTANT: in case you are not familiar with the **python cursor** you should check how things work before you start writing your analysis and quering the database, a quick full quide can be found here: **https://docs.djangoproject.com/en/dev/topics/db/sql/**.

Also in order to get a better understanding how the LHCbPR's database is structured and how to query information you can read the database section of the documentation and/or also take a look into the existing analysis modules (basic, timing) and take ideas out of them.

5.2.6 Using different templates

The default behavior of the functions is: the render(**kwargs) function uses the render.html with the data of the dictionary you return and the analyse(**kwargs) function uses the analyses.html with the data of the dictionary you return. You can **override** the default by specifying a **template key** in the returned dictionary with the different template you want to use. For example in the **timing analysis** we have 3 different ways of visualizing the data: TreeGrid, TreeMap, singleLevel so we have different analyse templates. So in case we want to use the analyseTreeMap.html(a different template we created in the templates/timing/ directory) instead of the default analyse.html template in the analyse(**kwargs) function we do :

```
return {
          #....
          #here reside the data we need in our template
          #and here we override the template to be used:
          'template' : 'analysis/timing/analyseTreeMap.html'
        }
```

Also there are some templates for general use in the templates/analysis directory like the error.html, so in case an error occurs with your data of with the database inside the analyse(**kwargs) or the render(**kwargs) function then you can do:

```
if Error:
  return {
           'template' : 'analysis/error.html'
           'errorMessage' : 'My errorMessage accessed by {{ errorMessage }}'
         }
```

This way you are able to create as many templates as your needs to diplay different messages, visualize in different ways your analysis results by overriding the template variable in the returned dictionary.

5.2.7 Bookmarking, trigger results, errorChecking

The template ,which your render.html, inherits provides **automatic bookmarking** with the following functions:

- prefillAll();
- prefill(); //per object

The prefillAll() will take the values from the url(GET request data) and will prefill the elements by id(keys of GET dictionary), or the prefill() will just perform the previous action for a specific values(also check the **trigger()** in the example code). So if you want to enable **bookmarking** in your render.html you can have something like this:

```
$(document).ready(function() {
 /* in case you want to prefill a single element you can
  * you can use this jquery function .prefill() :
  * $("#my_element_id").prefill();
  */
  //or else you can just call prefillAll();
  prefillAll();
  /* this method will take the values from the url(GET request data)
   * and will prefill all the available elements with the specified values
  */
 /* also if you want to automatic retreive the results in the
  * bookmarking url call trigger in your code :
  trigger();
 /* this will "click" the retreive results if the url contains trigger=true
});
```

Finally if you want to make **errorchecking** to your elements' values before the ajax request from the "Retreive results" button you can define a function called: **checkRequestData(requestData)** in your script block in the render . Let's assume that in your analysis page you want to always have at least two options selected before you press the button to retreive the results, in case the user doesn't select less than two options then an error message should be printed with the problematic field and the reason of the error, example code :

```
function checkRequestData(requestData){
//the requestData dictionary contains the data
//that will be sent to the analyse(**kwargs) in the format
//we previously mentioned
  //initial empty error object
  var errors = {}

  var myoptions = requestData["options"].split(",");
  if (myoptions.length < 2)
    errors["options"] = "At least two options must selected!";

 return errors;
}
```

The **checkRequestData(requestData)** method will automatically be called , before the ajax request to the corresponding analyse(**kwargs) function of the analysis module, passing the request data to it. If the returned errors' object is empty then the ajax request will be performed and the analyse.html template will be loaded to the page, if the errors object is not empty then a dialog will pop up printing the problematic fields (keys of the errors' object) along with the reason (values of the keys) and the ajax request won't be performed.

In our example we create an empty error object. If the user doesn't select at least two options we add an entry to the object error['options'] , as key we give a name for the problematic field(can be any word not necessarily the element id) and as value we give the reason.

5.2.8 Adding extra functions to the analysis

Let take the case you need from your render.html to perform an extra asynchronous request with your analysis module before you retreive the results, (for example the attributes filtering by group in the basic analysis).

Let say that you want an extra function to hello_analysis module called getMyStuff, then you have to define a method in your module like:

```
def getMyStuff(**kwargs):
  #the kwargs here contain again the request and requestDat
  # as the analyse(**kwargs) function
  #your function's code from here:
```

To access your new extra method through ajax you must make the request to the following url:

```
url = "{{ ROOT_URL }}analyse/hello_analysis/functions/getMyStuff/"+appName;
```

The **ROOT_URL** is returned to all LHCbPR templates so just use it the way it is, the same with the **appName** variable. So here follows the format of the url to access your extra function(change only the bold words) :

url = "{{ ROOT_URL }}analyse/**name_of_your_module**/functions/**extra_function_name**/"+appName;

5.2.9 Logging analysis

For logging in your analysis module you can use the django loggers (python logging module), here follows an example:

```
import logging

#important get the analysis_logger
logger = logging.getLogger('analysis_logger')

def render(**kwargs):
    logger.info('I am inside the render function!')
    #
    #...
    #
```

Using the **analysis_logger** from django (check the loggers in the LHCbPR's settings.py file) will save the logs in the LHCbPR's logs directory under the file: **LHCbPR/django_apps/static/logs/analysis.log**.

For more info on django logging: **https://docs.djangoproject.com/en/dev/topics/logging/**

5.2.10 Accessing ROOT through subprocess in LHCbPR

LHCbPR production release is deployed on a shared apache server. In the machine where the service is hosted ROOT is not installed thus we can not (and do not want to) access ROOT modules through pyROOT straight from the LHCbPR django process. For that reason when we want to use ROOT in LHCbPR we open a **subprocess** (from the main process) in which we communicate with a function (a custom function we write ourselves) which can import and use pyROOT.

The main process communicates, exchange messages/data, with the subprocess (we can also call it child process) through a **pipe**. In order to make the communication easier we wrapped the **pipe** inside a class called **subService**. The **subService** class can be found the following python file:

vim **LHCbPR/django_apps/tools/viewTools.py**

The subService class provides the following **methods**:

- **connect**: opens a subprocess calling the **ROOT_service.py** (will be explained later)

- **send**: sends data (any data, objects etc) to the subprocess (ROOT_service.py)
- **recv**: receive data from the subprocess
- **finish**: does nothing, was kept for legacy reasons (compatibility with another subService class)

The subService class opens a subprocess when the user calls the **connect** function. The subpService calls the ROOT_service.py in a subprocess passing the needed environment so the ROOT_service.py can access pyROOT, this python file can be found:

vim **LHCbPR/django_apps/tools/ROOT_service.py**

So the user can access any python function from the **ROOT_service.py** file through the subService class.

Ok now let's assume that from your analysis module you want to access pyROOT (eg in order to handle a ROOT file which contains histograms). First you need to write your **python function** which will handle your data using ROOT and place it inside the **ROOT_service.py** (check other function inside ROOT_service like basic_service etc). Once you have your function inside the ROOT_service.py you can access it through the subService class from your analysis module. Your main process (analysis module) will communicate with the ROOT_service through the **send** and **recv**(receive) methods of the subService class. Here follows an example showing how to write a "remote" function and access it through the subService class.

Let's assume that your remote function (inside the subprocess) will take two numbers, will calculate their sum and return it back to the father proccess. Here is the function:

```
#your function must take a remoteService object (subService instance)
#in order to communicate with the father process
def calculate_sum(remoteService):
    #receive the first number from the father process
    first_number = remoteService.recv()

    #get the second number
    second_number = remoteService.recv()

    #calculate the sum
    sum = first_number + second_number

    #send the answer back to the father process
    remoteService.send(sum)
```

Place the above function inside the ROOT_service.py then **very important** and **mandatory** place an entry inside the functionList dictionary (a dictionary inside the ROOT_service.py file) passing as key the name of the function and as value the function itself(like a pointer to the function), shown as below:

```
functionList = {
    'histograms_service' : histograms_service,
    'basic_service' : basic_service
    #any other function may exist here...
    #here add your new added function passing the name
    #of the function as key and value pass the function
    'calculate_sum' : calculate_sum
                }
```

Once you have done the above you ready to access your "remote" function from your analysis module, here follows an example:

```
#import the subService class
from tools.viewTools import subService

#
#...
#

def analyse(**kwargs):
    #
    #...
    #

    #define the two numbers to calculate their sum
    number1 = 10
    number2 = 20

    remoteservice = service()
    #open the subprocess
    #in case any error occurs return an error, example
    if not remoteservice.connect():
        #logger.error('Could not connect to remote service')
        return { 'errorMessage' : 'Connection with remote service failed!',
                 'template' : 'analysis/error.html' }

    #IMPORTANT
    #once you call connect
    #define which method(function) from the subService you want to use
    #pass to the send method the name of your function as string
    remoteservice.send('calculate_sum')

    #once you call your method you can start communicating with it
    #send the two numbers
    remoteservice.send(number1)
    remoteservice.send(number2)

    #then receive the sum the child process (subprocess)
    my_number_sum = remoteservice.recv()

    #...
    #
```

That way you can write methods which can have **access to pyROOT** (remember the ROOT_service.py can import ROOT). The above way to access ROOT in LHCbPR may seem a little inconvenient but it is the only way for the moment. In the future(soon enough) when the **root.js**, accessing ROOT functions will become much more easier without the need to run ROOT on a local machine or communicating with other processes (as we do now).

6 Handler development

In this section we will show how to write and test a new handler to collect your data.

6.1 Getting started

First get the latest **LHCbPRHandlers** project from LHCb GIT repository:

git clone /afs/cern.ch/lhcb/software/GIT/LHCbPRHandlers.git

Then make sure you are using at least **python2.6** to write and test your handler python module. Once you have python2.6 and a local copy of the LHCbPRHandlers project you ready to start.

6.2 How to write handler

Each handler is python module which **extends** the **BaseHandler**(it resides in the **LHCbPRHandler/handlers** directory) and works as a parser which collect data. The only difference from any usual parser if the way it saves the attributes.

The **BaseHandler** class provides the following methods:

- saveInt
- saveFloat
- saveString
- saveFile

Each of the above methods take 4 arguments

- **name** : the name of the attribute you save(example : 'cpu_execute_time')
- **data** : the data/value your attribute has (example : 12.45)
- **description**(optional) : any description you want to add for the attribute (example : 'this number represents...')
- **group**(optional) : the group(if any) in which the attribute you save belongs(example : 'Timing')

Let take for example the following case:

```
execution_time = 12.45
#If you want to save the number above you can save it like:

# remember the 4 arguments: name, data, description, group
self.saveFloat('execution_time', 12.45,'the total execution time', 'timing results

#or just save it without description/group, just its name and its value/data:
self.saveFloat('execution_time', 12.45)
```

Remember to use the right method for the corresponding **type**, if you want to save a string use the saveString function if you want to save an Integer use saveInt etc

You can also save **files**(eg .root files) by calling the **saveFile** function giving a 'name' for your file attribute and a 'filename' which must be the actual path to the file you want to save. For example:

```
#lets say that you have a file: my_results_file = '/afs/cern.ch/path_to/my_file'
## attribute name  path to file
self.saveFile('my_results_file' , '/afs/cern.ch/.../path/my_file' )

#also you can add a group and/or a description(like the explained above)
```

The handler python file must have the same **name** as the **handler class**, For example if your handler class is called 'TimingHandler' then the python file must have the name 'TimingHandler.py'

Second you must have an __init__ where you call super(self.__class__, self).__init__() (as shown in the example below)

At last you must override the method collectResults(self,directory). When a handler is invoked after a job execution , the script which invokes the handler passes to it a directory which contains the job results. So any file your handler needs must be found it in the given directory (the directory argument of collectResults function). In other words the directory argument will provide the path of the directory where the output of the job reside. You can there access the files by their names. Usually the **stdout** of the job is saved in a file under the name **run.log**

Here follows an example of a handler called "TestHandler":

```
#we need it just to pretty print the example
import pprint

#import the BaseHandler
from BaseHandler import BaseHandler

class TestHandler(BaseHandler):

  def __init__(self):
    #it is needed here
    super(self.__class__, self).__init__()

  def collectResults(self,directory):
    #here are some attributes we want to save
    a_float_number = 12.46
    an_int_number = 800
    a_string = 'This is my string'

    #here we save the above attributes
    self.saveFloat("a_float_number", a_float_number,
      description = "This is float's description(optional)",
      group = "MyGroup(optional)");
    self.saveInt("an_int_number", an_int_number,
      description = "This is my int's description(optional)",
      group = "MyGroupInt(optional)");
    self.saveString("a_string", a_string,
      description = "This is string's description",
      group = "MyGroup(optional)");

#and here you can test it by hand
#before you submit it
if __name__ == "__main__":
  #create an object
  test =  TestHandler()

  #  CHECK Note(1) after the code example
  # in real handlers you must provide
  # a directory where your results reside
  # and then the collectResults method
  # can access your files
  test.collectResults('some_directory')

  #then print the collected results:
```

```
myprint = pprint.PrettyPrinter(indent=4)
myprint.pprint(test.getResults())
```

Important, when you add a new handler in the repository remember to add a new entry(the name of the module) in the Handler table in the database in order to see/be able to select the new handler in the web interface. If you do not have the permissions to do it yourself ask the administrator to add it for you.

Note(1) the **__main__** function of the example is recommended for testing purposes. While the **collectRunResults** will be called by the LHCbPR system, the __main__ will allow you to check your handler's behaviour before commiting it.

Note(2) There also another way to test your handler. First go the **http://lhcb-pr.web.cern.ch/lhcb-pr/jobDescriptions/** , choose your application, then choose your job description for which you want to test your handler and generate the script for local run (by pressing the corresponding button in the web iterface of the job description details dialog). Once you save your auto-generated script make the following changes in order to properly test your new handler:

Edit the **script** as shown below(read the comments)

```
#COMMENT out ALL the lines until the JOB_DESCRIPTION_ID
#
JOB_DESCRIPTION....
#In the HANDLERS variable add the name of your class, in our example
#it should look like this
HANDLERS="TestHandler"
#
#LEAVE the rest of the code as is
#
#then COMMET OUT the following line:
#python -c "import os,urllib,zipfile;urllib.urlretrieve....

#then at last edit the line:
#python LHCbPRHandlers/collectRunResults.py.... etc
#provide here the path to your local LHCbPRHandlers instance
#and REMOVE the -a argument(it is the last argument),
```

Once you edit your script as explained above, make the script executable:

chmod +x path_to_script

Execute it and check if any error occurs in the stdout/stderr of the shell(where you execute the script).

7 Maintenance

In this section is described how to maitain the LHCbPR project.

Currently the production release path is:

/afs/cern.ch/lhcb/software/webapps/LHCbPR

There exist the LHCbPR project, the .htaccess file, the lhcbpr.fcgi etc.

7.1 myconf.py file

The **myconf.py** file can be found in the LHCbPR project: **LHCbPR/django_apps**

Here is a sample of a myconf.py file, the comments document the variables:

```
#proper values can be found in the current production myconf.py file
dbname = 'name_of_the_database'
dbuser = 'username_for_the_database'
dbpass = 'database_password'

key = 'key_django_needs'

#ATTENTION
# LOCAL must ALWAYS False in order the shibboleth to be enable
#the only case is can be True is for local deployment and testing of the
#LHCbPR project
LOCAL = False

#DEBUG should be False for production
#in case of local deployment you can have it true
DEBUG = False

#ATTENTION - VERY IMPORTANT
#This variable must contain the root url of project on the hosted server
#for example currently the LHCbPR is served on a shared afs apache
#under the url cern.ch/lhcb-pr/, in that case the ROOT_URL must be:
ROOT_URL = '/lhcb-pr/'

#In case the deployment machine is changed also the ROOT_URL must change
#for example if you setup a VM with an apache (eg alamages.cern.ch VM) and the roo
#of your project is alamages.cern.ch/ the ROOT_URL must be:
#ROOT_URL= '/'
#and so on in other cases

#another example :
#in case the project is served under cern.ch/myothersite/
#then the ROOT_URL must be:
#ROOT_URL = '/myothersite/'
```

7.2 Cronjob pushing the results in database and DiracStorageElement

7.2.1 Overview

Let have an overview of how the results are pushed to the database

After a job execution the handlers are called (through the **collecRunResults.py** script in **LHCbPRHandlers**) and they produce a zip file containg the gathered data (each zip file has a UUID name). As soon as a zip file is produced it is uploaded to **DiracStorageElement** (either automatically via collectRunResults, check –help of the script, or manually by the user) using the **sendToDB.py** script from the LHCbPRHandlers.

The uploaded zip files in DiracStorageElement, are pulled from a **cronjob** and get pushed in the database. The cronjob is run by the **lhcbpr** service account in **acrontab** on lxplus. The cronjob checks if there is a new uploaded zip file, it checks if the UUID name of the zip file exists in the **AddedResults** table in the database. If it does not exist in the AddedResults table that means it's a new zip file and so it pushes it in the database.ssh lxvoadm

7.2.2 DiracStorageElement

DiracStorageElement name : **StatSE**

In order to connect to the storage space of StatSE via a terminal and not only through the DiracStorageElement API:

1. Connect to lxvoadm machine : **ssh lxvoadm**

2. Once you are in the lxvoadm connect to the volhcb25 machine: **ssh volhcb25**

In **volhcb25**, the DiracStorageElement's StatSE root folder is the **/data**. The zip files are uploaded inside the **/data/uploaded** folder. The cronjob checks the uploded folder for new zip files. When a new zip file is pushed in the database it is moved from the **uploaded** folder to the **added** folder (/data/added).

So the two folder used in StatSE (volhcb25 machine) :

1. **/data/uploaded** folder : this is where the new zip files containing the results are uploaded.

2. **/data/added** folder : this is where the zip files are moved after they have been pushed in the database. This folder's zip files can de deleted depending on the administrator's will. As soon as the zip files are pushed in the database it be useless keeping them since their data is in the database now. But we keep them in case an error occurs while pushing a zip file, you can check the logs and find the UUID of the zip which caused the problem, get the problematic zip file from the added folder and investigate what went wrong (if the log file's exception message is not enough).

7.2.3 Cronjob pushing the results

The AFS cronjob is running under the **lhcbpr** service account. To check the acronjobs you do:

acrontab -l (for more information about managing afs crontab please visit:

http://services-old.web.cern.ch/services-old/afs/afsguide.html and check the "Cron Jobs" section)

You will probably get this output :

***/5 * * * * lxplus.cern.ch /afs/cern.ch/lhcb/software/webapps/LHCbPR/LHCbPR/django_apps/pushtodb**

So the **lhcbpr** service account runs a cron job (executing the corrensponding script: pushtodb) every 5 minutes on a lxplus node

7.2.4 Refresh Dirac Proxy (LHCb certificate), Cron Job

The **lhcbpr**'s service account cron job uses LHCbDirac to access DiracStorageElement and pull the uploaded zip files. In order to access DiracStorageElement we need to have a proxy which we obtain once we run lhcb-proxy-init (for more info on lhcbpr certificates check:

https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/Certificate) Usually when you call **lhcb-proxy-init** it generates a proxy in the **/tmp/** on the machine where you called it. **BUT** in our case the afs cron job runs every 5 minutes on a **different** lxplus node so it won't locate the generated proxy on the /tmp/ (because it might be on a different node where we call the lhcb-proxy-init). So we need to provide a **custom path** to the **lhcb-proxy-init** command about where to generate a proxy. When you call lhcb-proxy-init it checks the current SHELL for a **X509_USER_PROXY** variable, if it is not set then it generates the proxy to the /tmp/ folder, but if it is set it generates a proxy using the variable value (**X509_USER_PROXY**'s value)

Refreshing Dirac proxy of cronjob:

1. **ssh lhcbpr@lxplus.cern.ch** , login to lxplus as lhcbpr service account

2. **echo $X509_USER_PROXY** , make sure that X509_USER_PROXY is set, if not set it manually :

   export X509_USER_PROXY=/afs/cern.ch/user/l/lhcbpr/private/myProxyFile , currently this line is added to **lhcbpr**'s **~/.bashrc** file so there is no need to set it again. In case you change the path to generate the proxy just update also the .bashrc with the above export command with the new path.

   IMPORTANT In case you change the current path (/afs/cern.ch/user/l/lhcbpr/private/myProxyFile) also change the:

   /.../LHCbPR/django_apps/lhcbPR/management/commands/pushToDB.py

   change the 10th line, and you new path:

   ```
   #set custom path for the proxy
   os.environ['X509_USER_PROXY'] = '/afs/cern.ch/user/l/lhcbpr/private/myProxyFi
   ```

3. **lhcb-proxy-init -v 900:00** , call lhcb-proxy-init by providing how long you want your proxy to be valid (in this example will be 900 hours valid)

4. **acrontab -l** , finally (optional step) check if the cronjob exists in the crontab , you should get the following output:

   ***/5 * * * * lxplus.cern.ch /afs/cern.ch/lhcb/software/webapps/LHCbPR/LHCbPR/django_apps/pushtodb**

   if you don't get the above output, edit the acrontab (**acrontab -e**) and add the above line.

7.2.5 Checking the added results, checking the logs for errors

All the logs of the LHCbPR exist in the LHCbPR/django_apps/static/logs.

You can check the logs of the cron job by checking the following files:

1. **cd /afs/cern.ch/lhcb/software/webapps/LHCbPR/LHCbPR/django_apps/static/logs**
2. **tail check.log**, shows when the cron job checks the DiracStorageElement for new uploaded zip files (a new entry every five minutes). In case it finds a new zip file, a new entry in the log is added with the zipfile's UUID.
3. **tail pushToDB.log**, a line for each zipfile added in the database, contains info: if the pushing of the zip file was successful or not. In case of an exception the exception is logged into the log file.

Note: In case you want to see if your results where pushed successfully in the database just search the UUID of your zipfile in the **pushToDB.log** and check whether your zipfile was added successfully or not.

7.3 Managing the files

By default, Django stores files locally, using the MEDIA_ROOT and MEDIA_URL settings

(variables located in LHCbPR/django_apps/settings.py) For more info about django managing files check the online documentation :

https://docs.djangoproject.com/en/1.5/

Currently the way we handle the uploaded files is the following :

Using the django FileField field in our ResultFile model (LHCbPR/django_apps/lhcbPR/models.py) we have a relative path to the saved file. The relative path has this format: **job_description_id/job_id/filename**. So in order to get the full path of your file first you get the relative path from the FileField and then using MEDIA_ROOT (in the setting.py)you get the full path :

MEDIA_ROOT/job_description_id/job_id/filename

Currently the MEDIA_ROOT path of the files is :

/afs/cern.ch/lhcb/software/webapps/LHCbPR/data/files/

So in case you want to move the files to another afs directory just take the files (the relative paths in the database will remain the same) and move them to new directory, then **change** the **MEDIA_ROOT** variable with the new root directory in the:

LHCbPR/django_apps/settings.py

7.4 Managing Static files (js, css etc)

All the static media of the LHCbPR project exist in:

LHCbPR/django_apps/static directory. There you can store the js, css files or any other static file you need for the project (in the corrensponding existing folders, or create new folders etc).

Django keeps the **STATIC_URL** in the LHCbPR/django_apps/settings.py file. In case you want to change your static path edit the settings.py and change the **STATIC_URL** variable.

A proper way to access your static files through your html templates in the use the **STATIC_URL** variable instead of hardcoding the path:

Example of accessing your static files from your html templates:

```
<link rel="stylesheet" href="{{ STATIC_URL }}css/lhcbPR_new.css" media="screen">
```

That way you can easily change your static path, where you store your static files, by just changing the STATIC_URL variable in the settings.py file, **without** having to also change all the paths of your templates.

Note: for more info of how django manages the static files check the django's online documentation

7.5 Making a new release, deploy on other machine

First and most important thing, your **LHCbPR/django_apps/** directory must contain a valid **myconf.py** file (for more info check the corresponding documentation section), example of this file can be found in the current production release directory.

Let's assume that you have worked on a local copy of the GIT repository of the LHCbPR project and you want to make a new release and update the current production one:

1. **cd /afs/cern.ch/lhcb/software/webapps/LHCbPR/LHCbPR/**
2. **git pull**, to fetch the latest changes in the LHCbPR's code
3. **touch ../lhcbpr.fcgi**, to refresh the lhcbpr.fcgi file on the remove apache server
   (path of lhcbpr.fcgi : /afs/cern.ch/lhcb/software/webapps/LHCbPR)

IMPORTANT:

In case you deploy on new machine or change the folders in the static etc, make sure that you give to the **afs web server** (on any other server you use to deploy the project) **permissions to write** the following folders (if you forget that the server will crash each time it tries to write on these folders)

- **.../LHCbPR/django_apps/static/timingJson**, path to save the CSV files for the timing analysis
- **.../LHCbPR/django_apps/static/images/histograms**, path to save the generated histogram.png images
- **.../LHCbPR/django_apps/static/logs**, path to save the log files

In order to give write permissions to the afs webserver :

1. **cd .../path/to/folder/**, go to the folder you want to give write permissions
2. **fs sa -dir . -acl webserver:afs rlidwk**
3. **fs la**, finally check if the permissions are right (permissions must contain **webserver:afs rlidwk**)

IMPORTANT:

In case you deploy the LHCbPR project in a new machine **do not forget** to change the **ROOT_URL** of **myconf.py** in the :

LHCbPR/django_apps/ folder (for more info about the **myconf.py** check the corrensponding Maintenance section of this documentation file).

7.6 Managing the logs

The logs of LHCbPR exist in the **LHCbPR/django_apps/static/logs/**. In case you want to change the loggers you can edit the **LHCbPR/django_apps/settings.py** and change the **LOGGING** python dictionary. For more info of Django logging can be found in the Django official documentation.