A RESTful Web Service Interface to the ATLAS COOL Database.

S. A. Roe, on behalf of the ATLAS Collaboration

CERN, CH-1211 Genève 23, Switzerland

E-mail: shaun.roe@cern.ch

Abstract. The COOL database in ATLAS is primarily used for storing detector conditions data, but also status flags which are uploaded summaries of information to indicate the detector reliability during a run. This paper introduces the use of CherryPy, a Python application server which acts as an intermediate layer between a web interface and the database, providing a simple means of storing to and retrieving from the COOL database which has found use in many web applications. The software layer is designed to be RESTful, implementing the common CRUD (Create, Read, Update, Delete) database methods by means of interpreting the HTTP method (POST, GET, PUT, DELETE) on the server along with a URL identifying the database resource to be operated on. The format of the data (text, xml etc) is also determined by the HTTP protocol. The details of this layer are described along with a popular application demonstrating its use, the ATLAS run list web page.

1. The COOL Database

The COOL database [1] used by ATLAS represents both a schema and API which present a folder-like view of data to the user as illustrated in Fig. 1. Each folder has a fixed payload of named variables of specific types (float, string etc.) and each channel in that folder stores one payload. The folders may in turn be 'tagged', allowing different version of data to be stored and retrieved (for use in real data taking or simulation, for example). Folders are components of 'foldersets', which creates a file-system like path to the data in the database. In Fig.1, the high voltage data for ATLAS Semiconductor Tracker (SCT) modules can be found in the /SCT/DCS/HV folder, and both the current and voltage are stored as floating point numbers for each channel (corresponding to each SCT module). The data are inserted with an Interval Of Validity (IOV) representing the time or run/luminosity block numbers between which the data are valid.

The COOL API was written in C++ but has been incorporated into the Python language as a Python module (PyCool [2]). The API is designed to be database technology neutral: it can address Oracle, MySQL or SQLite databases, given the appropriate connection parameters.

2. Motivation for a Web Service Interface

Web pages are a natural and easy way to present information on detector status, and during 2006-2007, a number of web pages were written which presented information from the COOL database

© 2010 IOP Publishing Ltd

reflecting the detector status and operating conditions, both actual and past (in response to user input). Because the COOL API requires certain libraries to be dynamically loaded, the web servers would either a) run a shell script to invoke the appropriate environment before launching COOL *for each query* ; b) Eschew the use of the COOL API and use PHP/Perl/Python basic database libraries on the server ; c) Run periodic jobs which reproduced the database data required to local static files and then use those as the data source. These methods run the risk of being, respectively, a) slow, b) dangerous (through the use of non-optimized queries and transmission of database passwords), c) Inflexible and inconsistent with the current data in the database. With this in mind, the author decided to write a web service interface with the following design goals:

i) A central interface able to address COOL for data retrieval and updating
ii) Language neutral (usable from PHP, Java, C++, Python…)
iii) Using the COOL API (rather than basic database access methods)

The natural choice was to start from PyCool and write the application in Python, using XML for the data transport since XML parsers now exist in every commonly used computer language.

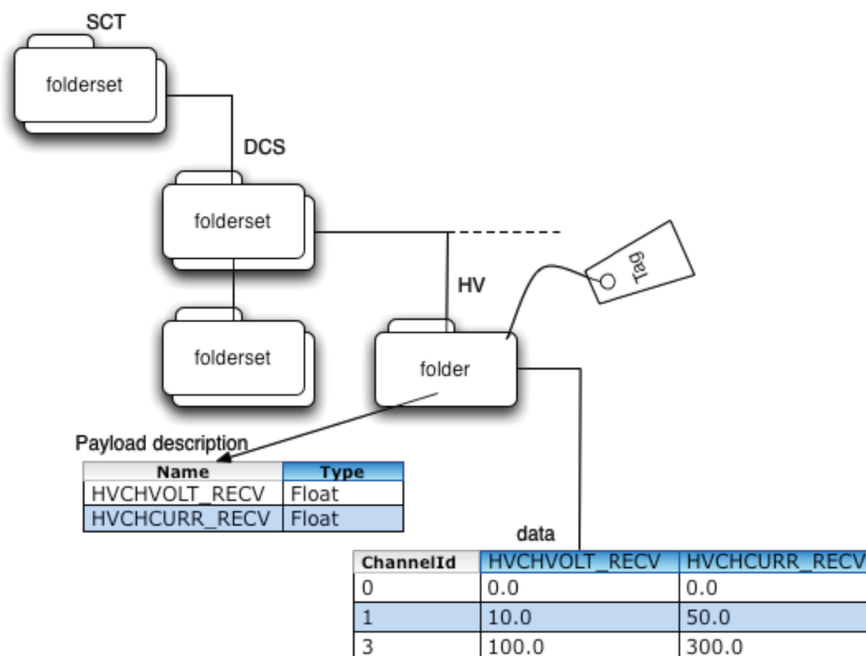

Fig.1: High voltage data for the SCT as seen in COOL

3. Web Service Design

3.1. Considerations

The explosion in web service technologies in recent years means that there is a broad choice of alternatives and languages which may be used on the server, and even within Python (given the existence of PyCool), it is possible to implement services as RPC (Remote Procedure Call) or SOAP (Simple Object Access Protocol) using available open-source modules, among other alternatives. Methods such as SOAP and RPC allow complex messages between the client and server, usually wrapped in XML syntax, and may maintain state between calls so that the actions of calls depend upon previous calls to the server. In the current design, the calls are necessarily constrained to the subset provided by the COOL API and they are 'atomic': data are retrieved or updated in a single transaction. These considerations, coupled with the hierarchically addressable nature of the data illustrated in Fig.1, led the author to consider designing a 'RESTful' interface as explained in the next section.

3.2. REST
Representational State Transfer, or REST, is a design philosophy for network based software architectures expounded in 2000 by Roy Fielding [3]. When applied to web services, this philosophy is often presented in the simplified form of practical interface details:

i) Each data 'resource' is identified with a unique URL (Uniform Resource Locator)
ii) The typical 'CRUD' operations (Create, Retrieve, Update, Delete) are implemented as HTTP Post (or Put, see below), Get, Put and Delete requests to the resource.
iii) The resource representation (HTML, XML, text…) is encoded as a MIME type in the HTTP header

Implicit in (ii) is the notion of statelessness of the application: each HTTP request contains all the information it requires to execute. (iii) may be redundant, since it assumes that the server can emit or receive different representations of the data, which may not be necessary. A web service adhering to these principles is said to be RESTful.

3.3. COOL URLs
The hierarchical nature of the COOL schema can be seen as a multidimensional coordinate system locating each datum in the database, and this location can be intuitively mapped to a unique URL representing the datum. As an example, consider the SCT high voltage status for a given module identifier:

Coordinate	Value
DB Server	ATLAS_COOLPROD
Schema	ATLAS_COOLOFL_DCS
COOL DB name	COMP200
Folder path	/SCT/DCS/HV
Timespan	2008-09-13:03:57:35 to 2008-09-13:04:12:00
Tag	HEAD
Channel	173357056

Table 1: Coordinates of SCT HV values

This can be mapped to a web service URL as:

http://servername.cern.ch/ATLAS_COOLPROD/ATLAS_COOLOFL_DCS/COMP200/SCT/DCS/HV/timespan/1221271020000000000-1221271920000000000/tag/HEAD/channel/173357056

The time is in ns of Unix epoch, a standard representation in COOL. The 'timespan' and 'channel' words act as data delimiters and visual cues. An HTTP Get request to this URL would retrieve the information, whereas a Put request would send a payload to the server to update the channel information for this IOV. To create resources (e.g. channels) there can be some confusion between the Put and Post methods; a Post method should be used when requesting a resource from the server, i.e. the server decides the actual resource location, whereas Put is used for creation of a resource where the resource location is indicated in the URL. There is some ambiguity when requesting the creation of multiple resources (e.g. many channels), and in this design the Post method is used on the parent resource (a folder) since the channel location is indicated by the payload and not in the URL. The actual payloads received and the definition of resources are described in the next section.

4. Implementation Details

4.1. CherryPy

CherryPy [4] is an open source Python web application server which enables one to expose Python classes and methods as URLs. E.g. given a class 'MyClass' with method 'helloWorld, a CherryPy application might expose this as the URL http://servername/MyClass/helloWorld, which would execute the method and send back the result. The servername and port are configured in a configuration file which is read by the application. CherryPy runs a multithreaded server so each request runs in its own thread, and can operate stand-alone or behind an Apache server. Many modules are available for additional facilities: caching, authorization, web templating, TWikis etc. In this application, particular use is made of the 'Routes' module which allows full control of the URL-to-class/method mapping, and the authorization module to enable password authorization. The Get/Post/Put/Delete methods on a resource can in principle be run on the same server in a single application, but the author decided to separate the functionalities on different server ports since the 'Get' is likely to receive the most traffic and may require separate monitoring.

4.2. URL Mapping

In addition to the channel data indicated in section 3.3, various other resources can be identified in COOL which deserve their own URLs. Some of these resources can only be retrieved, not manipulated. The Python code which maps these resources uses a ':variable' notation to extract a variable, and a '*multiple' notation to indicate a group of data which may contain the forward-slash. Static strings act as data delimiters. The mappings using this notation are in Table 2.

Resource	URL
List of nodes in db	/:server/:schema/:dbname
Folder payload	/:server/:schema/:dbname/*folderPath/payload
Folder description	/:server/:schema/:dbname/*folderPath/description
No. channels	/:server/:schema/:dbname/*folderPath/length
List channels	/:server/:schema/:dbname/*folderPath/list
List tags	/:server/:schema/:dbname/*folderPath/tags
Channel values	/:server/:schema/:dbname/*folderPath/timespan/:iov/tag/:tag/channel/:chans

Table 2: Resource to URL mapping

The extracted variables may be strings subject to further interpretation; e.g. the :chans variable may be a single number, a range (such as 0-100), a comma separated list or a wildcard asterisk. The creation of a new resource involves sending a resource definition, via Post, to the parent resource. Thus the creation of new folders means posting a folder definition to the /:server/:schema/:dbname URL, and new channels are created by posting to the /:server/:schema/:dbname/*folderPath/ URL.

4.3. Data Transport: XML

XML is a natural first choice for the data format, and the database structure maps naturally into an XML tree. A sample response from a Get request for channel values (in this case, the SCT low voltage status) is shown in Fig 2.

Journal of Physics: Conference Series **219** (2010) 042021 doi:10.1088/1742-6596/219/4/042021


```
<channels server="ATLAS_COOLPROD" schema="ATLAS_COOLOFL_DCS" dbname="COMP200"
folder="SCT/DCS/CHANSTAT" since="1226271600000000000" until="1226274125000000000"
tag="" version="single”>
 <channel id="138950656" since="1226270581000000000" until="1226274125000000000">
  <value name="LVCHSTAT_RECV">193</value>
  <value name="STATE">17</value>
 </channel>
 <channel id="138950656" since="1226274125000000000" until="1226277728000000000">
  <value name="LVCHSTAT_RECV">193</value>
  <value name="STATE">17</value>
 </channel>
</channels>
```


Fig.2: XML response to a data request

The XML is described by a W3C schema definition which may be found in [5]. The same format file may be used for updating channel values, by sending it to the database as a Put request payload. Similar definition files have been produced for the other Post and Put operations, allowing validation of the upload file before actually executing it, or creation of new files by hand in a schema aware editor such as oXygen[6] with predictive completion of text. In designing the XML format it was decided to avoid the use of XML namespaces as they are not widely used in our community and can lead to confusion. Using the same or similar formats for data retrieval and upload invites the possibility of creating and copying databases simply by downloading the definition from an existing database and uploading it to a different URL. Future versions are planned which use XInclude [7] syntax to import the XML definition from a Get request to one database into a Post/Put request to another, omitting the additional download step.

4.4. HTTP Headers and Responses

When requesting the XML data format, the HTTP ‘Accept’ header should be set to MIME type ‘text/xml’, and when executing a Put or Post, the ‘Content-Type’ header should be similarly set; this allows for the introduction of new representations (e.g. text/plain, application/json) in the future.

The HTTP server response code is also normally available to the user and may be set by CherryPy to indicate successful completion or an error condition. The HTTP response codes in use are indicated in Table 3.

HTTP Code	Produced by
201 (Resource Created)	Successful Post request
204 (No Content)	Successful Delete request
400 (Bad request)	Trying a Post or Put with no payload
404 (Not found)	The URL is not recognized
409 (Conflict)	Trying to create a resource which exists
500 (Internal Server Error)	Catch-all for general error condition

Table 3: HTTP Response codes issued

In addition, a successful Post or Put request sends back a link to the created/updated resource so it may be verified.

5. A Sample Application Using the Web Service: The ATLAS Run List

The COOL – CherryPy server was first put into production at the end of 2007 and was quickly followed by the introduction of run information web pages using the service. These pages incorporated a user query to retrieve run information within a run number range or on a specific date, information

which was stored in the COOL database. A Model-View-Controller architecture was adopted as shown in Fig.3, using AJAX (Asynchronous Javascript And XML) techniques to retrieve and insert the data to the web page, thus avoiding the typical 'blinking' of a page refresh associated with classic form submission techniques. The view in this case is the web page itself, and the controller is an embedded Javascript program which queries the local server application (in Python); this effectively acts as an interpreter and local proxy for communication to CherryPy.

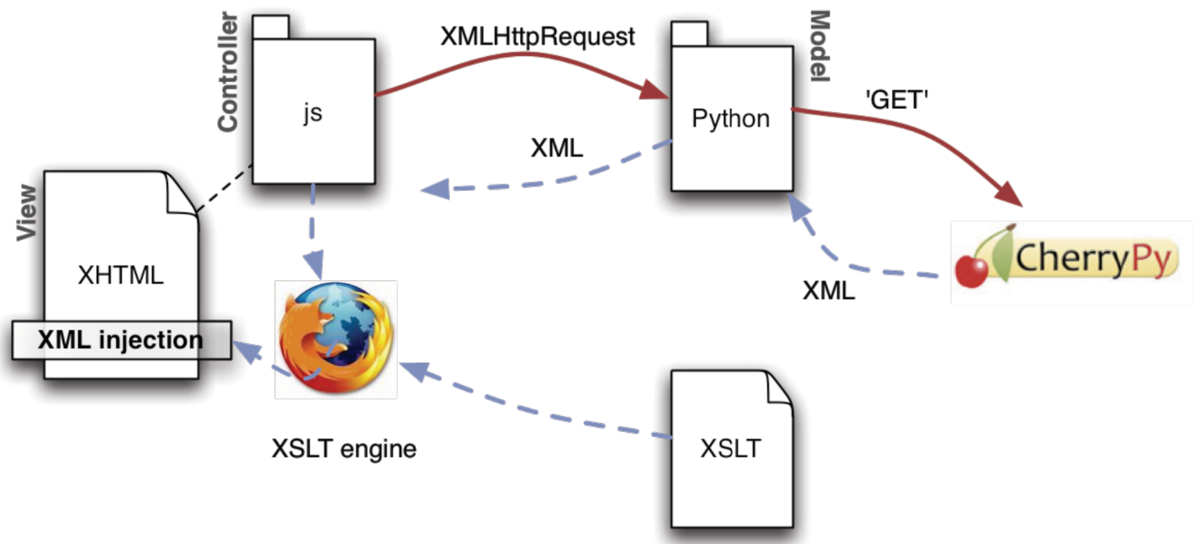

Fig.3: ATLAS run list web application architecture

The final (tabular) presentation of the retrieved data is executed in the browser itself by associating an XSLT (eXtensible Stylesheet Language for Transformations) stylesheet with the retrieved XML, thus the injected data remain in XML format and the view can be changed by altering the stylesheet alone. The XSLT also introduces hyperlinks to retrieved data, allowing further investigation of the run details. This application is still in use and being developed further.

6. Conclusion

Use of CherryPy in conjunction with an existing COOL Python API allowed rapid development of a uniform web service to query and manipulate the ATLAS COOL database without the installation of additional libraries, thus permitting additional languages such as Java to access COOL. In adopting a RESTful interface, the web service has become an intuitive portal to COOL which allows expansion to future data formats. It is currently used in many web applications where data are required from COOL.

Acknowledgements:
The CherryPy/COOL server is based on an original component written by Georg Stach, and the run list was based on software by Richard Hawkings. Andrea Formica, Else Lytken and others wrote applications based on the CherryPy layer and have helped with debugging. None of this would have been possible without the input of the COOL and PyCool developers: Andrea Valassi, Sven Schmidt and Marco Clemencic

References

[1] Valassi, A. et al. COOL, LCG Conditions Database for the LHC Experiments: Development

and Deployment Status, CERN-IT-Note-2008-019 and NSS 2008 Proceedings of the Medical Imaging Conference, Dresden, Germany.
COOL Package Documentation:
http://lcgapp.cern.ch/doxygen/COOL/COOL_2_6_0/doxygen/html/
[2] PyCool: https://twiki.cern.ch/twiki/bin/view/LCG/PyCool
[3] Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures, Ph.D. dissertation to the University of California, Irvine (2000)
http://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm
[4] http://www.cherrypy.org,
[5] Schema for channel upload: http://sroe.home.cern.ch/sroe/xsd/channels.xsd
[6] oXygen, Syncrosoft: http://www.oxygenxml.com/
[7] W3C Recommendation: XML Inclusions: http://www.w3.org/TR/xinclude/

Additional resources:

Richardson, L. & Ruby, S. *RESTful Web Services* (O'Reilly, 2007)
Hellegouarch, S., *CherryPy Essentials* (Packt Publishing, 2007)