A Python object-oriented framework for the CMS alignment and calibration data

Joshua H Dawes
on behalf of the CMS Collaboration, CERN
The University of Manchester
E-mail: joshua.dawes@student.manchester.ac.uk

Abstract. The Alignment, Calibrations and Databases group at the CMS Experiment delivers Alignment and Calibration Conditions Data to a large set of workflows which process recorded event data and produce simulated events. The current infrastructure for releasing and consuming Conditions Data was designed in the two years of the first LHC long shutdown to respond to use cases from the preceding data-taking period. During the second run of the LHC, new use cases were defined.

For the consumption of Conditions Metadata, no common interface existed for the detector experts to use in Python-based custom scripts, resulting in many different querying and transaction management patterns. A new framework has been built to address such use cases: a simple object-oriented tool that detector experts can use to read and write Conditions Metadata when using Oracle and SQLite databases, that provides a homogeneous method of querying across all services. The tool provides mechanisms for segmenting large sets of conditions while releasing them to the production database, allows for uniform error reporting to the client-side from the server-side and optimizes the data transfer to the server. The architecture of the new service has been developed exploiting many of the features made available by the metadata consumption framework to implement the required improvements.

This paper presents the details of the design and implementation of the new metadata consumption and data upload framework, as well as analyses of the new upload service’s performance as the server-side state varies.

1. Motivation for new Infrastructure
The Alignment, Calibrations and Databases group at the CMS Experiment[1] at the CERN LHC[2] rewrote a set of services during the first long shutdown of the LHC to respond to detector experts’ use cases, and redesigned the Conditions Database schema[3] (see figure 1). However, new use cases have required the writing of new services and rewriting/optimisation of existing services for Conditions consumption and upload. Details and analyses of these improvements are given in this paper. An extended version of the discussion in this paper (along with more implementation details) can be found in [4].

2. CondDBFW - Object-Oriented Conditions consumption framework
2.1. Introduction
The Python[5]-based services maintained by AlCaDB lack a common Conditions Metadata (the data stored in Conditions database schema, such as Tags and IOVs, as described in figure 1) consumption interface, where either Object-Relational Mapping frameworks or raw SQL queries
Figure 1: Diagram of the schema shared by all Conditions databases.

are used. Experts who maintain individual services should not be concerned with how to query for their data, rather with their own processing of the data.

CondDBFW is a Python-based framework that provides an interface for Conditions Metadata consumption in the form of the pattern of use in section 2.2, and abstracts away from SQL or ORM-based querying by making instantiation of CondDBFW objects (instances of ORM classes built using SQLAlchemy[6], representing Conditions Metadata rows) and selection from a Conditions database equivalent.

2.2. Pattern of Use

2.2.1. Obtaining a CondDBFW.querying.connection object For Python scripts, the CondDBFW.querying module provides the utility function querying.connect for constructing and setting up a connection object. For Python shell sessions, the CondDBFW.shell module provides the utility function shell.connect which tracks all of the connections made.

Connection objects are the starting point of every query the user will execute. Each has type CondDBFW.querying.connection and contains all of the connection information needed to abstract away from SQL or ORM-based querying; the user need only give the database connection string and, if credentials are required, a netrc file or dictionary that contains them.

2.2.2. Proxy Method usage No operations on Conditions Metadata can be performed without first obtaining a CondDBFW object, which is done by using methods defined on a CondDBFW.querying.connection object. This connection object has methods that will use its factory to instantiate an object, hence behave as a proxy. Invocation of each of these methods with search criteria returns a new CondDBFW object whose attributes are either populated by values from the Conditions database, or empty (or the returned reference is to a NoneType if no results were found), where the search criteria for the selection are defined by giving the value for a column to the column’s keyword argument in the proxy method (e.g, connection.tag(name="..."))

Additionally, the way in which a value given for a column is used in the query depends on the type given as the keyword argument defined by the proxy method. For example, an str will result in a straightforward equality, whereas classes such as CondDBFW.models.Range, CondDBFW.models.Radius and the Python list allow for querying on lists of values, and CondDBFW.models.RegExp allows for querying on regular expressions.

The value returned from a proxy method query is either a CondDBFW object, if a single
In [2]: connection = shell.connect("oracle://...", secrets="...")
In [3]: connection.tag(name="JDAWES_TEST_TAG_FOR_PLOTTING")
   .iovs().get_members("payload_hash").data()
Out[3]:
[u'9853a0866ebf253e8d2caa3be5cd296da13ce7f5', [...]]

Figure 2: Demonstration of the use of method chaining in CondDBFW to obtain the list of Payload hashes used by a Tag’s IOVs.

result was returned: a json_list object, if multiple results were returned, which wraps a Python list of CondDBFW objects and guarantees the definition of certain methods on data that CondDBFW uses; or None, if no results were returned. An empty CondDBFW object can be obtained by calling a proxy method with no arguments.

CondDBFW objects returned from queries have methods defined on them that exploit Foreign Key relationships. For example, CondDBFW Tag objects have the method iovs, which returns a json_list of iov objects contained within the Tag.

Depending on the result of the query, more Conditions Metadata can then be accessed by using methods defined on Conditions Metadata and json_data_node objects. As an example, figure 2 demonstrates how to get the Python list of all Payload hashes used by the IOVs in a Tag.

2.2.3. Writing to another Database
An object returned from a query using SQLAlchemy belongs to the session used to query for it, in that it contains information including the containing session and the object’s schema. To enable writing objects’ data to databases different to the ones they were read from, CondDBFW makes ORM objects session independent, that is, not belonging to any SQLAlchemy session, before their data is written.

2.3. Distribution
The Conditions Metadata framework is deployed through the CMS software collection (CMSSW)[7] and a separate repository[8] for use in the web services. CMSSW is used to provide an object-oriented tool for access to Conditions Metadata in the context of CMS offline data processing.

The main aim of the framework is to provide a uniform method of Conditions Metadata consumption for the CMS Python-based services, which are predominantly web-based, so there must be a web safe release pushed to the separate repository that the web services can use. A commit in the project history is marked as being web safe with a tag if that version of CondDBFW can be used inside the lifetime of an HTTP request.

3. CondDBFW Application: cmsDbCondUpload
3.1. Introduction
The improvements implemented in the new upload service for Conditions are given in this section. Multiple HTTPS requests and a retry mechanism are used (implemented with the HTTP(S) request class CondDBFW.url_query.url_query) to reduce vulnerability to unstable network connections, in response to the old service having occasional problems with large uploads (the definition of large depending on the network connection speed). Homogeneous error reporting from the server-side to the client-side is provided, so the user has the option to see the traceback from the server-side if their upload failed. Additionally, the data sent over HTTPS is reduced to the minimum required at the time of processing by FCSR Filtering (see section 3.3.3) and Hash Checking (see section 3.3.4).
This section is concluded by analyses of the performance of the new upload service as the server-side state varies.

3.2. Architecture and Deployment

3.2.1. Client-side

The client-side of the new upload service is in two parts: the upload script (which the user must run to perform an upload) and the CondDBFW.uploads module. CondDBFW.uploads contains all of the upload logic, whereas the upload script is responsible for passing data (see section 3.3) from the user and synchronising versions of CondDBFW between the client and server-sides.

This split into two parts allows the local version of CondDBFW (containing the CondDBFW.uploads module) to be updated to match the server-side’s version. For each execution, the upload script queries the server-side for the git commit hash it is using and pulls that commit if its hash is different to the local hash. If the upload script does not have write permissions in the working directory, a temporary directory is used for the duration of the upload.

The new service uses upload sessions to link the HTTPS requests in an upload together, controlled by a single instance of the CondDBFW.uploads.uploader class constructed for the upload. To make the service more maintainable, the methods defined on an uploader that communicate with the server correspond bijectively to routes on the server.

3.2.2. Server-side

The server-side is a web service written in Python using the Flask framework and a web safe version of CondDBFW, reducing the amount of code needed for database querying significantly and making it much easier to maintain.

For Metadata Uploading the server-side defines the class MetadataHandler, which validates Tags and IOVs during instantiation, hence a MetadataHandler instance will contain Conditions Metadata that has been validated with respect to the Upload Metadata, that is, the data given by the user either through command line arguments or an Upload Metadata file. If a file is given, command line arguments override the options in the file.

The new homogeneous error reporting is achieved by errors being converted into JSON string representations of dictionaries (containing the error message and traceback of the error) and returned to the client-side. The client-side CondDBFW module parses the JSON and informs the user of an error.

3.3. Upload Process

The upload process (see figure 3) is split into the distinct stages of Parsing, Initialising an Upload Session, FCSR Filtering, Hash Checking, Payload Uploading and Metadata Uploading.

3.3.1. Parsing

The parsing stage of the upload obtains the Upload and Conditions Metadata, where the Upload Metadata is obtained by the upload script and the Conditions Metadata by the uploader constructor.

A special case of the input data is an upload that does not use an SQLite database file. If a Tag already exists and a Payload is known to be already in the destination Conditions database, the user can append an IOV to the Tag by specifying the Payload hash to use, the IOV’s since and the destination Tag. In this case, the Upload Metadata is used to construct the Conditions Metadata, and the Tag is constructed locally by asking the server for the Tag information in order to allow the data sent to the server-side during the Metadata Upload stage (see section 3.3.6) to pass the Tag Match check.
Read in Conditions Metadata (Tag and Tag's IOVs).
Put Conditions Metadata and Upload Metadata into dictionary.
If using SQLite database file and either Tag or IOVs are not found, exit.
Begin upload - ask server for an upload session.
Preprocessing:
- Convert all sinces to Lumi-based.
- Set source since to first IOV since (in list ordered by since) if not given.
Was upload session given?
Was the --fcsr-filter flag given?
yes
no
No upload session, nothing can be done. Exit.
no
Take a list of hashes from the final list of IOVs.
Send hashes to server, and get back list of hashes not found in destination database.
Send payload of each hash not found.
Encode each BLOB as Base64 and send in body of request.
Finally, send Metadata dictionary.
This is a mix of Conditions and Upload Metadata.
Write logs - this involves finalising client-side log, and receiving server-side log in response from Metadata upload.
Report completion and log names to the user.
Use this list to check for hashes that were found on the server, but not found locally.

Figure 3: A State Diagram of the Upload Process performed by the new upload service.

3.3.2. Initialising an Upload Session  An upload session’s representation in the upload service is a row in the upload_sessions table in the usage schema (the subset of the Conditions schema that holds upload sessions) whose expiry_time column is greater than the current time by 30 seconds. This session spans multiple HTTPS requests to the server-side of the upload service. A Tag is locked to an upload session (hence cannot be written to by any other user) if and only if an upload session exists and is valid for that Tag.

Each upload session is uniquely identified by its authentication token, which is the sha1 hash of the JSON string representation of the dictionary containing the username and the timestamp of the creation of the upload session. A token’s expiry time is defined as the expiry time of its upload session.

The first request in a normal upload is to open an upload session and set an expiry time on it (5 minutes into the future), the response to which is the unique token for the new upload session in the usage schema if the upload session was successfully created. This token is used in all future requests in that upload, and tells the server-side which upload session each request belongs to. If the upload session could not be created (an upload session could already be open on the destination Tag), the user is informed and the upload process ends.

The last request is the Metadata upload request which closes the upload session (see section 3.3.6). Once the last request is complete, the upload session is considered closed.

The usage class found in the app.usage module is used on the server-side to carry out all authentication using either the first-time (username, password) pair, or the token corresponding to an upload session. This class also carries out all logging tasks, since the server-side maintains a separate log for each upload session.

3.3.3. FCSR Filtering  FCSR Filtering is a validation mechanism that processes IOVs on the client-side with respect to the First Condition Safe Run (FCSR) derived from the value of --fcsr-filter given. The First Condition Safe Run is the first run at which Conditions can be added for a given synchronization without overwriting frozen (already computed) Conditions. The synchronization given by --fcsr-filter only overrides the destination Tag’s synchronization
if that Tag has synchronization any or validation, or does not yet exist.

The IOVs are still validated by the MetadataHandler class on the server-side, given that the FCSRs of many synchronizations are independent of the Tag.

If a synchronization is given, an HTTPS request is sent to the server-side with the destination database connection string, the destination Tag (because some FCSRs are dependent on the Tag) and the synchronization value given by the user. If the destination Tag does not exist and the FCSR is dependent on the Tag, the FCSR is defined to be −1 since there are no IOVs.

FCSR Filtering is useful for a detector expert to export Conditions to a Tag that has not been accepted into a synchronization yet, as it allows validation of the IOVs in the Tag according to the intended synchronization’s FCSR. The checks on the server-side verify that the synchronization of a destination Tag is not overridden by the user if it is not any.

3.3.4. Hash Checking  Hash checking in the new upload service is a significant optimisation that starts by sending an HTTPS request to the server-side containing the list of hashes computed locally from the Conditions Metadata. The response to this request contains the list of Payload hashes that were not found in the destination Conditions database, computed on the server-side by only adding hashes for which the CondDBFW query for the Payload returned None.

Given that a list of hashes not found on the server-side is obtained, an additional process carried out at this stage is to determine a sublist of the original list sent to the server containing only hashes that are not found locally or on the server. This is done by marking these Payload hashes as not existing locally if the local SQLite database file does not contain the Payload when the Conditions Metadata is first read during the Parsing stage (see section 3.3.1).

An error is thrown if an IOV uses a Payload that does not exist on the client or server-side of the service.

3.3.5. Payload Uploading  Payloads are each uploaded in their own HTTPS request, with the Base64 encoded Payload BLOB in the request body. Separate HTTPS requests are used so that retries can be done on individual Payloads, making the upload less vulnerable to unstable network connections.

Once the Payload data has been received on the server-side, a checksum in the form of the sha1 hash is computed from the concatenation of the Payload’s data and object_type columns and compared with the hash sent from the client-side. If this recomputed hash matches the one sent, an attempt is made to write the Payload to the destination Conditions database and, if it doesn’t match, an error is returned to the client-side, which then retries the upload.

3.3.6. Metadata Uploading  The purpose of the Metadata Upload stage is to validate the Conditions Metadata with respect to the Upload Metadata, and then write the corrected data to the destination Conditions database. The process followed to perform the validation in the constructor of the MetadataHandler class is:

(i) If the destination Tag does not exist, it is created in the destination Conditions database with synchronization any, meaning the IOVs added to it are not currently restricted by any FCSR. The IOV validation described after this list is then applied for the synchronization any.

(ii) If the destination Tag does exist, the local Tag must first be checked for a match with the destination Tag, which requires that the Tags’ time_type and object_type properties match. If the check is passed, the IOVs must be validated by the algorithm presented after this list.

The algorithm for adjustment of IOV since values is as follows: first, the first IOV in the list ordered by IOV since values in ascending order must be less than or equal to the since given by
the user, or there will be a hole in Conditions coverage. Then, find the greatest IOV since that is less than or equal to the FCSR obtained (either from --fcsr-filter or the destination Tag’s synchronization). This since corresponds to the final IOV(s) (there could be multiple IOVs with the same since) whose range will include the FCSR, and so all IOVs with since values below the FCSR are modified to equal it.

4. Optimisations

The new upload service has been shown to significantly shorten upload times with a subset of uploads where the upload data can safely be reduced. Figure 4 is derived from logs for the two most common types of upload: when Hash Checking concludes that all Payloads must be sent, and when Hash Checking and FCSR Filtering are used to significantly reduce the amount of data sent, resulting in the clear difference in the two plotted lines.

![Figure 4: Plots showing the effect of the applications of Hash Checking and FCSR Filtering.](image-url)

For the upload that sent all of its Payloads, the plot denotes the point at which each Payload is written to the destination database by the meeting points of pairs of vertical and horizontal lines. For example, the plot between the second and third vertical lines (log lines 33 to 41) denotes sending a Payload of size $\approx 100$MB and is characterised by an initial large step, corresponding to the Payload being sent over HTTPS, followed by the server-side decoding the Payload data from Base64, and a final step corresponding to writing the Payload to the destination database.

From figure 4, the main cause of inefficiency in the upload process is the time taken for Input and Output, either done by sending data over a network connection or reading from a local SQLite database file. For example, the step in time at log lines 17 to 18 for the upload that sent 5 Payloads represents the time taken to read the Payloads from the local SQLite file, a process that cannot be optimised solely within Python. A similar problem occurs at log lines 39 to 40 which denote the step in time caused by writing to the destination Oracle database.
5. Conclusion
Several optimisations were implemented with the redesign of the upload service, including splitting each upload over multiple requests; giving the user the option to filter their IOVs by an FCSR on the client-side before sending Payloads; and only sending Payloads to the server-side that were not already stored in the destination database.

FCSR Filtering is not the strongest of the optimisations, however it can reduce the number of Payloads to be uploaded significantly and in a small amount of time (the plot for the upload that didn’t upload any Payloads shows that FCSR Filtering took $\approx 2$ seconds ranging from log line 6 to 13). However, its main use case is to allow detector experts to perform safe (guaranteed by the rules in section 3.3.3) early validation on IOVs depending on the intended target synchronization.

Hash Checking is always used because it does not require specific criteria. The usefulness of this optimisation is demonstrated by the plot in figure 4 where no Payloads are sent, hence the upload time is reduced by an order of magnitude.

Finally, CondDBFW has proven to simplify queries greatly in the upload service and the object-oriented design is extended into the upload service’s architecture. The framework has also proven to be a powerful tool for performing non-trivial operations on Conditions Metadata. Furthermore, the deployment mechanism used distributes the correct version of CondDBFW for the upload service without relying on the user to update manually.

Acknowledgments
I thank the members of the Alignment, Calibration and Databases group within the CMS Collaboration at CERN. I extend thanks to Andreas Pfeiffer, my supervisor, for selecting me; and to Giacomo Govi, Giovanni Franzoni, Gianluca Cerminara and Salvatore Di Guida for invaluable input into the development of the new Conditions Data framework.

I also thank my family, including my dad, Geoffrey Dawes; my brother, Simon Dawes; and my girlfriend Sarah Whitaker, all having provided every support I have needed while working in Geneva. Finally, I thank my mum, Cheryl Dawes, who sadly lost her battle with cancer during the writing of this paper. I dedicate my work at CERN, and all of my future work, to her - the most inspiring person I will ever know.

References
[1] The CMS Collaboration 2008 Journal of Instrumentation 3 S08004
[2] Evans L and Bryant P 2008 Journal of Instrumentation 3 S08001
[3] Di Guida S, Govi G, Ojeda M, Pfeiffer A and Sipos R 2015 Journal of Physics: Conference Series 664 042024
[4] Dawes J H 2016 A Python object-oriented framework for the CMS alignment and calibration data Tech. rep. URL https://cds.cern.ch/record/2220970
[5] Python Language https://www.python.org/
[6] SQLAlchemy http://www.sqlalchemy.org/
[7] CMSSW Github Repository https://github.com/cms-sw/cmssw
[8] CondDBFW Gitlab Repository (CERN) https://gitlab.cern.ch/cms-ppdweb/cmsCondMiniFramework
[9] Git Version Control System https://git-scm.com/
[10] Python Flask http://flask.pocoo.org/
[11] SQLite https://www.sqlite.org/