ConfigDB

MongoDB internally defines a database as a set of collections, with each collection consisting of a set of documents, each of which is essentially a JSON object. Additional python code has been written to use MongoDB as a configuration database. These classes and routines can be imported as:

```
from psalg.configdb.typed_json import *
import psalg.configdb.configdb as configdb
```

Typed JSON

JSON does not provide any type information for the values it contains. For a configuration database, this can be problematic, as some numeric values might be limited to integers or even a small set of integers. Therefore, additional information is added to the JSON configuration object to provide type information. Any field name ending in "**:RO**" is considered to be read-only and will not be displayed by the graphical editor. An additional key, "**:types:**" will be added to the top level dictionary. This key maps to a dictionary with roughly the same structure as the JSON object, except that instead of containing values, the dictionary contains type information. This information is either:

- A basic type: one of "**UINT8**", "**UINT16**", "**UINT32**", "**UINT64**", "**INT8**", "**INT16**", "**INT32**", "**INT64**", "**FLOAT**", "**DOUBLE**" or "**CHARSTR**".
- A string denoting an enumeration type.
- A list, the first element is either a basic or enumeration type, and the other elements are integers denoting array sizes.

The "**:types:**" dictionary also has an "**:enum:**" entry defining all of the enumeration types. The keys of the "**:enum:**" dictionary are the enumeration types, and the value is a dictionary mapping enumeration names to integer values.

It is also assumed that any lists in the JSON object contain objects of the same type, so the "**:types:**" dictionary does not contain a list at that level but the type information for every element of the list.

A few top-level keys have predefined types, reflecting their use in configuration objects. These reserved keys are:

- "**detType:RO**", a mandatory **CHARSTR**.
- "**detName:RO**", a mandatory **CHARSTR**.
- "**detId:RO**", a mandatory **CHARSTR**.
- "**doc:RO**", an optional **CHARSTR**.
- "**alg:RO**", an optional dictionary containing:
  - "**alg:RO**", a mandatory **CHARSTR**.
  - "**doc:RO**", an optional **CHARSTR**.
  - "**version:RO**", a mandatory ["**INT32**", 3].

The cdict Class

In order to simplify the creation of typed JSON objects, the **typed_json** module defines the **cdict** class. The constructor for this class takes an optional argument which is either an instance of another **cdict** or a dictionary representing a typed JSON object that is used to initialize the new **cdict**. The main methods for this class are:

- **setInfo(detType=None, detName=None, detId=None,** doc=None)
  Set the top-level reserved keys to the specified values.
- **setAlg(alg, version=[0,0,0], doc="")**
  Set the top-level reserved "**alg**" key to a dictionary containing the arguments.
- **set(name, value, type="INT32", override=False, append=False)**
  The "**name**" parameter is a "flattened" name with dot-separated fields. Each field is used as a dictionary key or list index in turn to describe a particular value in the JSON object. (As a bit of syntactic sugar, list indices may be separated from the previous field by an underscore or a dot. That is "**a.6.b**" or "**a_6.b**" are equivalent.) This routine sets the referred to value to the **value** parameter with the specified **type**. If the hierarchy already exists and has a different type, an exception is thrown unless **override** is **True**, in which case the hierarchy and type is overwritten. In the event that **name** refers to a list, **append** controls whether the list should be overwritten or appended to.
- **typed_json()**
  Return a dictionary representing a typed JSON object (with the "**:types:**" key, etc.).

- **get(name, withtype=False)**
  The "**name**" parameter is a flattened name as described for the **set** method. The current value of this name in the hierarchy is returned. If **withty pe** is **True**, values that are basic types will return a tuple, the first element of which is a string which names the basic type and the second element is the value.
- **getenumdict(name, reverse=False)**
  If **name** is not a defined enumeration type, return **None**. Otherwise, return a dictionary with the enumeration mapping. If **reverse** is **False**, the mapping is from names to integers, and if it is **True**, the mapping is fromintegers to names.
- **define_enum(name,value**)
  This method defines a new enumeration type, **name**. **value** is a dictionary mapping names of the enumeration type to integer values.

The **typed_json** module also has a few helper functions to deal with typed JSON dictionaries.

- **getType(dict, name)** returns the type of the value referred to by the flattened **name**, throwing an error if **dict** does not have any such value.
- **getValue(dict, name)** returns the value referred to by the flattened **name**, throwing an error if **dict** does not have any such value.
- **updateValue(dict, name, value)** stores a new **value** into the typed JSON **dict** referred to by **name**. **value** is always a string. Numeric values are converted, and array values are space-separated. 0 is returned on success, and non-zero values indicate an error.

The configdb Class

Configuration management with MongoDB is handled by the **configdb** class. In general, we add additional documents to the database, but do not change existing ones, so we can keep a complete history of configuration changes. The basic organization is:

- Each type of device has a collection that includes possible configurations for instances of this device. The configurations are stored as typed JSON objects.
- The configuration database includes a set of **hutches**, one collection per hutch.
- Each hutch has a set of **aliases**, which describe a specific running condition (for example: "**BEAM**" or "**NO_BEAM**").
- The hutch collection has a set of documents which include an alias, a version number ("key"), and a list of dictionaries (one per device). These dictionaries include the name of the instance of the device, the device type, and an ID which identifies the particular configuration document in the device type collection that should be used. The highest version number for a particular alias is the current configuration for this alias.

The constructor for the **configdb** class has the form **configdb(server, h=None, create=False, root="NONE")**, where:

- **server** is a string identifying the MongoDB server and is either "**user:password@host:port**" or "**host:port**".
- **h** is a string identifying the the default hutch.
- If **create** is **True**, create any necessary DB entries.
- **root** is the name of the database to use. The DAQ configuration will be kept in "**configDB**".

The methods in the **configdb** class are:

- **set_hutch(h, create=False)**
  Set the default hutch to **h**. If **create** is **True**, create any necessary DB entries.
- **add_alias(alias)**
  Add a new alias to the default hutch.
- **add_device_config(cfg)**
  Add a new device type collection named **cfg**.
- **get_hutches()**
  Return a list of all defined hutches.
- **get_aliases(hutch=None)**
  Return a list of all aliases for the specified **hutch** (or the default hutch if the parameter is **None**).
- **get_device_configs()**
  Return a list of all device types.
- **get_key(alias, hutch=None)**
  Return the highest version number for the **alias** in the specified **hutch** (or default hutch if **None**).
- **get_devices(key_or_alias, hutch=None)**
  Return a list of devices in the specified **hutch** (or default hutch if **None**). **key_or_alias** specifies the particular configuration to examine: if it is a string, use the current configuration for this alias and if it is an integer, use the configuration with the specified version number.
- **modify_device(alias, value, hutch=None)**
  Modify the current configuration for the specified **alias** in the specified **hutch**. **value** is a typed JSON dictionary where the **detName:RO** field is the name of the device and **detType:RO** is the device type. This raises an exception if there is an error and returns the newly written version number otherwise.
- **get_configuration(key_or_alias, device, hutch=None)**
  Get the configuration for the specified **device** in the specified **hutch** (or the default if this is **None**). **key_or_alias** specifies the particular configuration to examine: if it is a string, use the current configuration for this alias and if it is an integer, use the configuration with the specified version number ("key"). Internally "key" is used for all lookups by the server-side code below.
- **transfer_config(oldhutch, oldalias, olddevice, newalias, newdevice)**
  Copy the current configuration for device **olddevice** with alias **oldalias** in hutch **oldhutch** to the current hutch configuration for with alias **newalias** device **newdevice**.
- **get_history(alias, device, plist, hutch=None)**
  Retrieve a history for the list of flattened names in **plist** for the specified **device**, **alias**, and **hutch**. The return value is a list of dictionaries, each one with **date**, **key**, and all of the **plist** elements as keys.

Design

Mike Browne designed this based roughly on Matt's configdb design document: https://docs.google.com/document/d/12BlCMCWGy3X9Z9QEZn9AtkjVipYA0_x3ZiyfDYrRo-c/edit. Some things were changed, but this was the starting point. Chris Ford worked with Murali Shankar on the backend http database development.

Code

The server-side code (which receives http requests on the database server) is here: https://github.com/slac-lcls/psdm_configdb. The client-side code (which forms the http requests) is here: https://github.com/slac-lcls/lcls2/tree/master/psdaq/psdaq/configdb.

Configuration history management

Two configdb commands can be used to manage configuration history:

- **configdb history**: this command can be used to explore the configuration history of a certain device. It takes a hutch/alias/device or hutch/XPM/xpmname string as argument (plus all the usual configdb options: --user, --password, --url-, etc.). For example:

  ```
  configdb history --user tmoopr --password <usual> tmo/BEAM/hsd_0
  ```

  The command returns a list of configurations, each with date (in UTC time zone) and "key" entry. The "key" can be used to retrieve the specific configuration using the configdb rollback command.

- **configdb rollback**: this command can be used to retrieve a specific device configuration and make it the current configuration for the device. It takes a hutch/alias/device or hutch/XPM/xpmname string as argument (plus all the usual configdb options: --user, --password, --url-, etc.). Additionally, it requires a --key argument. For example:

  ```
  configdb rollback --user tmoopr --password <usual> --key 266 tmo/NOBEAM/hsd_0
  ```

  The command will not write the retrieved configuration to the database unless the --write option is used,(it will just print it to the screen, together with a warning), so this command can also be used as a viewer for historical configurations (without the --write option)

Configuration History and Keys

DISCLAIMER: The following are just deductions. I (Valerio) have no direct insight on how configuration keys work.

- Keys are like commit in a git repository: they are snapshot of the configuration of all devices in a hutch at a certain point in time. Much like modifying a file generates a new repository state (a "commit") the same does modifying the configuration of a device
- Each new change of the configuration of the device in a hutch (irrespective of the alias, BEAM or NOBEAM), generates a new key (key numbers are integers increasing with each configuration change)
- Continuing with the git metaphor, aliases (BEAM, NOBEAM, etc.) are like branches in a repository

Mike Browne writes: I think Valerio's comments above are pretty spot on. The integer is essentially a version/commit number. It uniquely identifies everything at a particular moment in time... you change stuff, you get a new key/number. We're saving a complete history, with the idea if it gets too much at some point, someone in the future will write a history-pruning tool. 🙂

Two Suggested New Tools

Merging Schema Changes

There is a working example of schema update in psdaq/psdaq/configdb/hsd_config_update.py

issue: we don't handle merging of schema changes to the config objects

ric's figurative proposal:
- script to read db and write json file
- modify ts_config_store.py to put the new schema in dbase
- third script applies settings from json file to the dbase ignoring the ones
  that are dissimilar
(get_config.py has some of the tools to do this)

cpo proposal:
- one script (configdb_modify.py) that reads the configdb json, modifies it (with python code)
  and writes it back (a plug or a minus: we no longer have a script that knows the official
  state of the schema: the database itself is the source of truth).
  - if we ever lost the database we no longer know the configdb schemas

Another proposal (from Ric):

- Change the *_config_store.py scripts to take another command line argument that selects between creating a new configdb entry or modifying an existing one
    - If creating, behave as now
    - if modifying:
        - read the configdb entry into a json string (get_config.py: get_config())
        - apply the modified schema coded elsewhere in *config_store.py to configdb
        - use the json string from above to update the values of the new configdb entry (get_config.py: update_config() ??)
            - elements that are in the configdb entry but not in the json string are ignored
            - elements that are not in the configdb entry but are in the json string are ignored
- The RO elements like `help` and version numbers should be governed by the new schema code and not by the previous DB contents
- Delta configs (e.g. CALIB configdb entries) may need special handling to avoid turning a delta config into a base config and vice versa
    - Delta configs are recognized through the presence of a `_cfgTypeRef` element

item 1:

ts_config_V1:
config[group][0][rate]=10Hz
[1][rate]=100Hz
[2][rate]=100Hz

need a procedure to produce:

ts_config_V2:
config[group][0][rate]=10Hz
[2][rate]=100Hz

or

ts_config_V2:
config[group][0][rate]=10Hz
[1][rate]=100Hz
[2][rate]=100Hz
[4][rate]=100Hz

Rename/Remove Old Configdb Objects

ability remove and hide (in the control gui) detector configurations that aren't needed anymore
(hide: prepend an "_"?, rename timing_1 to _timing_1?)
rename = delete+create

Ric added: Maybe this should be thought of as **archiving** unneeded ConfigDb objects and we should therefore provide a way to *unarchive* them for when archiving is done erroneously or we change our mind. Need to consider how to handle the case when, for example, timing_1 is archived to _timing_1 and then a new timing_1 is created.

Mike Browne implemented the following rename/delete functionality. Murali is implementing a web interface on top of this:

```
c2 = cdb.configdb(mdb.server, "SXR", create=True, root=dbname)
c2.add_alias("FOO")                                      # 0
c2.transfer_config("AMO", "BEAM", "evr0", "FOO", "evr3")  # 1
with pytest.raises(Exception):
    c2.transfer_config("AMO", "BEAM", "evr0", "FOO", "evr3")
print("Configs:")
c2.print_configs()
cfg1 =  c.get_configuration("BEAM", "evr0")
cfg2 = c2.get_configuration("FOO", "evr3")
# These should be the same except for the detector names!
del cfg1['detName:RO']
del cfg2['detName:RO']
assert cfg1 == cfg2
c2.rename_device("evr3", "evr7", "FOO")
cfg3 = c2.get_configuration("FOO", "evr7")
# These should be the same except for the detector names!
del cfg3['detName:RO']
assert cfg2 == cfg3
with pytest.raises(ValueError):
    c2.remove_device("evr6", "FOO")
c2.remove_device("evr7", "FOO")
with pytest.raises(ValueError):
    cfg4 = c2.get_configuration("FOO", "evr7")
```

Rename/Remove Discussion

with Chris Ford, Murali, Matt, cpo on April 17, 2024

Murali highlighted some important questions with the new rename/remove features. Here they are, with answers from the group. I believe the conclusions were unanimous:

- do we want to update all old keys or only the latest key?
    - the consensus was that updating only the latest key did a better job of preserving the history (e.g. for comparison with xtc values). there was a weak counter-argument that the "configdb rollback" command would not work as naturally (would need to know both the old name and the new name).
- the detector name (e.g. trigger_0) shows up twice: once as a database-lookup key ("device") and once in the document with typed-json ("detName:RO", which is useful for converting to xtc). Do we want to edit both?
    - Yes, it is more natural to edit both. Matt pointed out that the daq code may overwrite detName:RO, but we weren't certain, and in any event it feels more elegant to have it be correct in the database
- do we want to create a new key with the changes?
    - Yes, because this does a better job of preserving the history
- do we want to create a new document with the new detName:RO?
    - Yes, because this does a better job of preserving the history
- what do we do if the user tries to rename to name that already exists in the current key?
    - We should raise an error, which would force the user to explicitly remove the existing conflicting name before doing the rename.

Removal is more straightforward: a new key will be created with the appropriate device removed, but history will be preserved in the old keys.

These answers will require changes to Mike Browne's code which Murali has kindly agreed to do.

CLI Overview

```
$ configdb -h
usage: configdb [-h] [--url URL] [--root ROOT] {cat,rm,cp,mv,history,rollback,ls} ...

configuration database CLI

positional arguments:
  {cat,rm,cp,mv,history,rollback,ls}
    cat                 print a configuration
    rm                  remove a configuration
    cp                  copy a configuration (EXAMPLE: configdb cp --create --write tmo/BEAM/timing_0 newhutch
/BEAM/timing_0)
    mv                  rename a configuration (EXAMPLE: configdb mv --write tst/BEAM/timing_45 timing_46)
    history             get history of a configuration
    rollback            rollback configuration to a specific key
    ls                  list directory contents

optional arguments:
  -h, --help            show this help message and exit
  --url URL             configuration database connection
  --root ROOT           configuration database root (default: configDB)
```

Configdb Server Logs

The configdb server runs on a subset of psdm[01-04], where multiple nodes are used in a "load balancer" manner. You have login to each of these nodes and look for logfiles in /u1/psdm/logs/configdb* to find the one that contains the transaction you are interested in.

Sample curl Commands

From Murali (for the dev configdb).

```
Get configuration for a device
curl -s -u "tstopr:passwordremoved" "https://pswww.slac.stanford.edu/ws-auth/devconfigdb/ws/configDB
/get_configuration/tst/BEAM/trigger_0/"

Rename device
curl -s -u "tstopr:passwordremoved" "https://pswww.slac.stanford.edu/ws-auth/devconfigdb/ws/configDB
/rename_device/tst/BEAM/trigger_0/?newname=trigger_1"
```

```
Remove device
curl -s -u "tstopr:passwordremoved" "https://pswww.slac.stanford.edu/ws-auth/devconfigdb/ws/configDB
/remove_device/tst/BEAM/trigger_0/"
```