`autologbook` lab database utilities

Michael P. Mendenhall

January 19, 2016

Contents

1 Introduction

`autologbook` is a set of utilities designed to help in common lab tasks: data logging and monitoring, equipment configuration, "checklist" routine monitoring forms. Information is

stored in sqlite3 (http://www.sqlite.org) database files, a highly robust and widely-used embedded database system. Various Python3 scripts facilitate reading and writing the database information, including a suite of CGI scripts providing an HTML web interface.

The autologbook utilities are structured around two independent databases, organized for different types of information. The "logger" database is intended for series of timestamped readings automatically logged from instruments, alongside textual annotations of lab events. The data can be monitored and plotted through a web interface. The "configuration" database is for storing configuration information for laboratory equipment (such as sets of voltages for power supplies), along with a mechanism for generating and archiving forms such as "shift change checklist" records.

2 Data logger

2.1 Database structure

The logger database schema is defined in logger_DB_description.txt. This is organized into "instruments" (describing initial sources of data), "readouts" (describing one particular aspect of an instrument providing a numerical readout), and "readings" (timestamped values for a readout). An additional table of of "log messages" holds timestamped textual notes, whether automatically generated (such as run start/stop annotations from a DAQ) or hand-entered.

2.2 Interface scripts

2.2.1 Logger server

DB_Logger.py provides a local server for database interactions, providing database query and modification functions through an XMLRPC interface. Two separate server processes are launched with read-only and read/write access to the database, permitting database write functions to be restricted to processes on the same machine (localhost network), with potentially broader access for data queries. Currently, both read and read/write servers are run on localhost only, until a reasonable system for broader but still restricted network access is implemented.

The logger server also provides a "data filter" mechanism for thinning out frequent readouts to a smaller number of points to be saved (either by simple decimation to one in every n, or recording points when (user-defined) "significant" changes occur).

2.2.2 Data sources

Data sources, capturing readings from instruments and sending them to the server, will need to be written for specific experimental applications. Example "fake" data sources are provided in `TestFunctionGen.py` and `FakeHVControl.py`, demonstrating how to set up instruments/readouts and send readings to the server process.

2.2.3 Web monitors

The read/query features of `DB_Logger.py` can be used by other applications to provide near-real-time monitoring and visualization of experimental conditions. The `web_interface/` directory contains various `CGI` script utilities for interacting with the databases, divided between `cgi-logger` for the logger database and `cgi-config` for the config database. `HTTPServer.py`, when run from inside the `web_interface/` directory, launches `Python`'s built-in simple `CGI` server for the scripts. The `cgi-logger` utilities include:

- `LogMessages.py`: recent text log messages
- `currentstatus.py`: summary of newest readings from each readout, with links to plots
- `plottrace.py`: plot recent time history of a readout (using `SVG` output from `gnuplot`)
- `HVArray.py`: example of a more customized status visualization for a $2 \times 4 \times 4$ PMT HV array

2.3 Tutorial walkthrough

Tell `Python` where to find `autologbook` programs:

```
export PYTHONPATH=<... path to ...>/autologbook/:$PYTHONPATH
```

Start up the logger server. From the `autologbook` base directory,

```
./DB_Logger.py --readport 8002 --rwport 8003
```

This sets the database server to provide a read-only connection on `localhost:8002`, and a read/write connection on `localhost:8003`. This process needs to be left running for everything else to work. If no previous `loggerDB.db` file exists, a new one will be created from the `logger_DB_description.txt` schema.

Feed readings into the database. For a real application, you will write your own code that provides this. For testing purposes, there are a couple of simulated data source processes to populate the DB (providing examples of how your own code would talk to the server):

```
./TestFunctionGen.py --port 8003
```

generates 5-minute and 12-hour periodic sine waves, and

```
./FakeHVControl.py --port 8003
```

simulates readings from 32 high-voltage channels.

Start the web server for viewing the logger data:

```
cd web_interface
../HTTPServer.py --host localhost --port 8000
```

launches an `HTTP` server visible only on your computer's internal (localhost) network at port 8000. In a web browser, visit `http://localhost:8000` to see the server output, showing the `web_interface/index.html` page. From here, you should find links to other view applications, such as the "current status" most recent readings (with plot links for the history of each), a logbook messages list (including messages created by the fake data generators), and a grid view for the (fake) PMT HV signals.

3 Configurations and forms

3.1 Database structure

The config database schema is defined in `config_DB_description.txt`. A configuration is identified by a name and a "family," which can group together multiple configurations with a common function (such as different choices of HV settings for a group of PMTs). Each named configuration is then associated with a set of key:value pairs. A "configuration history" table can be used to record when a particular configuration was applied (and queried to determine which configurations were in use at a particular time).

3.2 Interface scripts

3.2.1 Python interface

`configDBcontrol.py` provides basic convenience functions for creating and querying configuration sets.

`ConfigTree.py` "interprets" specially-formatted configuration sets into a tree structure: key names are treated as '.'-delimited "paths" in a nested structure, *e.g.* a "flat" configuration structure

$$x.a = 1, \quad x.b.u = 2, \quad x.b.v = 3$$

is expanded to a "nested" structure:

$$x : \begin{cases} a = 1 \\ b : \begin{cases} u = 2 \\ v = 3 \end{cases} \end{cases}$$

Specially formatted values (beginning with "@") are interpreted as links to (copies of) trees defined elsewhere, with "cascading inheritance" modifying values in the linked tree. For example, a new configuration for y linked to x with modifications could have a "flat" form of

$$y = @x, \quad y.a.q = \text{“Hello!”}, \quad y.a.r = \text{“Goodbye!”}, \quad y.b.v = 5$$

which would be expanded to:

$$y : \begin{cases} a = 1 : \begin{cases} q = \text{“Hello!”} \\ r = \text{“Goodbye!”} \end{cases} \\ b : \begin{cases} u = 2 \\ v = 5 \end{cases} \end{cases}$$

Here, y starts from the structure of linked x, but we have changed the value of $x.b.v = 3$ to $y.b.v = 5$, and added new sub-values q, r under $y.a$ — which may both hold a value of its own ($y.a = 1$), and contain sub-values q and r.

3.2.2 Web interface

The `web_interface/cgi-config/` directory contains `CGI` scripts for interacting with the configuration database through a web browser.

`ConfigWebManager.py` generates pages to browse, create, copy, and modify configuration families and configurations (in a simple "flat" view without interpreting any tree structure).

`Metaform.py` interprets/expands the tree structure of configuration objects for viewing and editing. Additionally, `Metaform.py` pays attention to specially-named keys in the tree to control how the tree is "rendered" to `HTML` output. For example, by including an "`!xml`" sub-branch:

$$x = \text{“Hello!”} : \begin{cases} !xml = \text{“div”} : \begin{cases} \#class = \text{“foo”} \\ \#style = \text{“color:mauve”} \end{cases} \end{cases}$$

x will be rendered to `HTML` as `<div class="foo" style="color:mauve">Hello!</div>`. Including a "`!list`" sub-branch causes an object to render those of its sub-branches starting

with '#', ordered by name. For example,

$$x : \begin{cases} !xml = \text{“ul”} \\ !list = \text{“li”} \\ \#a = \text{“first”} \\ \#b = \text{“second”} \end{cases}$$

will be rendered as the `HTML` list '`<ul> <li>first</li> <li>second</li> </ul>`,' appearing in the browser similar to:

- first
- second

Using the `XML` formatting directives, the web interface can be used to build web forms that submit information back to the database. By using links, these web forms can be quickly built out of "reusable" components, and copied from a blank "master copy" form to be filled out in multiple copies. The repository comes with a sample "starter" database (`configDB_starter.db`) filled with useful pre-defined form components and an example web form, which will automatically be copied into place when the code is first run (when no prior configuration DB is defined).

3.3 Tutorial walkthrough

Tell `Python` where to find `autologbook` programs:

```
export PYTHONPATH=<... path to ...>/autologbook/:$PYTHONPATH
```

Start the configuration DB webserver on localhost port 8001:

```
cd web_interface
../HTTPServer.py --host localhost --port 8001 --mode config
```

which will also automatically create an initial `configDB.db` if none previously exists.

Now you can visit the configurations editor from your web browser at `http://localhost:8001/cgi-config/ConfigWebManager.py`. The starting page for "Configuration set families" should come with a few families and objects pre-defined to try out, and you can add your own new configuration families from this page.

Follow the link in the list of families to see the objects in the family (which you can select and delete from that page — delete all to remove the family).

Following the links for a configuration object in a family brings you to the "flat" object editor mode, showing all the key/value pairs defined for the object. You can update, delete, rename, or create new parameters here. New configuration objects are created by copying an old one (field on bottom of page).

3.3.1 Tree-mode editor

To switch to the "tree view" viewer/editor, click the "(tree view)" link near the top of the "flat" editor page. This brings you to a "view" mode, rendering the object according to its contents (and special `!xml` keys). Click the "edit" link to switch to the tree-mode editor view.

Here, you will see a "Modify parameters" box showing a table view of all the keys defined at this level of the tree. The leftmost column shows the key names, starting with "(this)" for the value of the node being edited, and followed by each sub-node of "(this)". Names followed by an "@..." are links to a (copy of) another object; click the "@..." to view/edit the original linked object (beware: these changes appear every other place this object is linked!).

The next column of the table shows the `HTML` rendering of each sub-node. Following this is a column of text entry fields for updating the value at each node when the "Update" button is pushed, a column of "Edit" links to expand/edit nodes that have their own tree of sub-nodes, and a column of checkboxes to select items to delete with the "Delete" button. Note that some nodes may not have a value defined at this level of the tree structure, thus cannot be deleted here. This is the case for nodes with sub-nodes but no value assigned to "(this)" (which will be deleted when all sub-nodes are deleted), or for nodes "inherited" from linked values that have not been modified in this tree. "Deleteable" values defined at this level have their keys highlighted in yellow in the leftmost column; values not defined here have their "Update" text box highlighted in yellow, indicating that a new (deletable) entry will be added.

New key/value entries can be added in the "Add new parameter" box. You may enter '.'-separated names to define a sub-nodes. Entering a value of "@" will assign a value of "None" to a key (note, a node with value of "None" is distinct from a node with no value). Other values starting with "@" will be interpreted as links.

3.3.2 Forms example

The configuration database comes pre-loaded with an example of how the tree editor can be used to create web forms writing to the database. Starting from the configuration families

page, go to the "filled" family, and the "example filled" object; click "(tree view)". You are looking at an example of a web form with filled-in fields. Editing the field contents and clicking the "Submit" button will load the new contents to the database. Now, follow the "(edit)" link to see how this form was built.

The top node of the form is "(this) @form:example," which is a link to the blank "template" for this form without filled-in values. The form has a "!list = None" entry, causing it to appear in "view" mode as the series of its "#" keys, and an "!xml" entry defining the top-level `<form>` tag. The "#" keys place the various fields (constructed from sub-objects including "@obj:formfield" entry fields), along with the "Submit" button at the end.

Return to the simple "(flat)" editor via the link in the top title. Note that the only values defined in the database for this object are the "@form:example" link and the values typed in to particular fields — all the other structure is "inherited" from the linked template form.

You can create new empty copies of the form to fill in simply by creating new (empty) objects and setting "(this)" to "@form:example."