
AT-TPC DAQ Documentation

Release 1.0

Josh Bradt

Mar 27, 2017

1	Contents	3
1.1	Overview of DAQ system	3
1.2	Installation and initial setup	5
1.3	Configuring the system	11
1.4	Logging information about runs	13
1.5	Operating the DAQ system	14
1.6	Developer documentation	20

This manual describes how to install, set up, and use the web-based GUI for the AT-TPC's DAQ system. The documentation is divided into a few different sections. For some background information about the system, see [Overview of DAQ system](#). The [Installation and initial setup](#) section describes how to install the system and its dependencies, like Docker. The next two sections, [Configuring the system](#) and [Logging information about runs](#), show how to set up the system for data taking.

The most important section of this manual for experimenters taking shifts is probably [Operating the DAQ system](#). It describes how to configure the CoBos and start and stop runs. It also has instructions on how to record parameters about the runs, like pressures and voltages.

At the end of the manual is the [Developer documentation](#) section, which contains information about how the system is implemented. This is probably only of interest to people who want to maintain the system or add new features.

Contents

Overview of DAQ system

The AT-TPC DAQ is based on a collection of programs provided by the GET collaboration. These provide the back-end of the system by handling CoBo configuration and data recording. This web application serves as a front-end for those programs.

GET software components

There are two programs, in particular, that need to be running for each CoBo. They are:

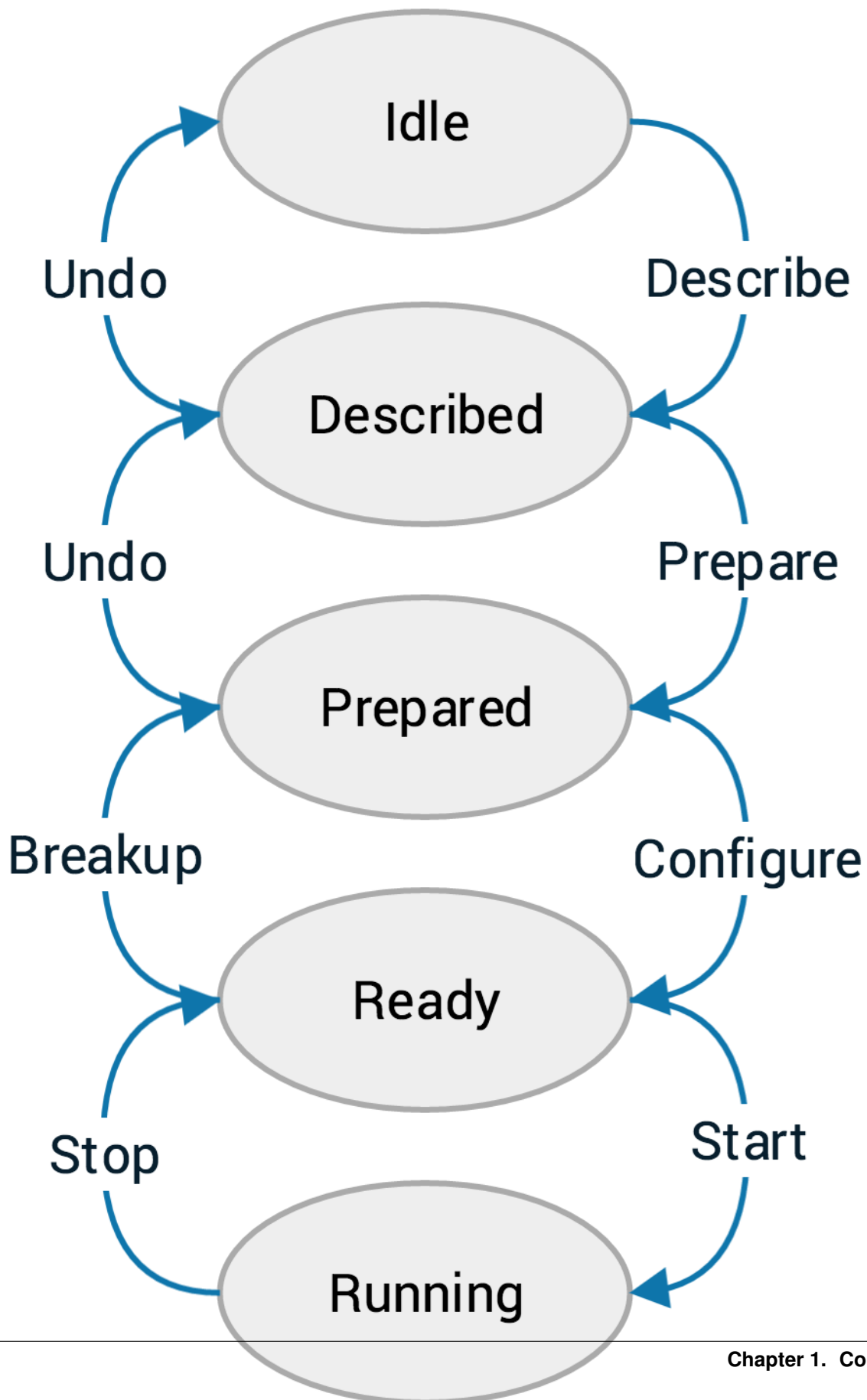
getEccSoapServer This program controls the CoBo. It sends the configuration to the CoBo and tells it when to start and stop acquisition.

dataRouter This program records the data.

The web interface controls the system by acting as a client for the getEccSoapServer. It does not communicate with the dataRouter directly.

CoBo state machine

The ECC server controls the CoBo using the model of a state machine. This means that that CoBo can be in one of several well-defined *states*, and to change from one state to another, it will undergo a well-defined *transition*. The state machine for the CoBo looks like this:



The ellipses represent the different states that the system may be in, the arrows along the right side show the forward state transitions, and the arrows along the left show the reverse state transitions.

Config files

Each forward transition requires a particular part of the configuration file. This is why we have three config files for each setup (or, alternatively, two true files and one symbolic link). The expected files are named:

- `describe-[name].xcfg` for the describe step
- `prepare-[name].xcfg` for the prepare step
- `configure-[name].xcfg` for the configure step

These names will be shown stripped of their prefix and suffix in the DAQ interface. For example, a file called `describe-cobo0.xcfg` will be shown as simply `cobo0`.

Web-based GUI

The interface to the system is a web application written in Python 3 using the [Django web framework](#) for the back-end with [Bootstrap](#) providing the front-end. The structure of the code is described briefly in [Developer documentation](#) and in comments directly in the code itself.

As Django apps can be a bit tricky to serve, the app has been structured to run inside [Docker](#) containers. The Dockerized version of the app can be built using the Docker compose utility like this:

```
docker-compose build
```

Installation and initial setup

Requirements

GET software

As mentioned [previously](#), this DAQ software depends on two of the programs from the GET software suite: the `getEccSoapServer` and the `dataRouter`. These programs are not provided with this package, so they must be compiled and installed separately before this package can be installed.

Docker

Docker and the Docker Compose tool are required to get the containerized version of the DAQ software running. Docker can be downloaded and installed from [its developers' website](#) or from your package manager if you're using Linux.

Networking

If you'll be running the system on multiple computers, be sure to consider where files will be stored. The ECC server will expect to find config files locally wherever it's running, so if multiple ECC servers are running on multiple computers, you will likely want to share a folder on your local network to keep the config files in.

Source code

Finally, get the latest version of the DAQ software from GitHub:

```
git clone https://github.com/attpc/attpc-daq.git
```

Always use the latest version from the Master branch. The version in the Develop branch may not be stable.

Creating the environment file

There are a few environment variables that need to be set to system-dependent values inside the Docker container. Several of these variables provide encryption keys or passwords, so this environment file is not in the Git repository (and it should *never* be committed to the repository!).

Create a file in the root of the repository with the following values. Remove the comment strings (starting with #) before saving it.

```
DAQ_IS_PRODUCTION=True           # Tells the system to use the production settings,
↳ rather than debug.
POSTGRES_USER=[something]        # A user name for the PostgreSQL database. Set it to
↳ something reasonable.
POSTGRES_PASSWORD=[something]    # A secure, random password that you will not likely
↳ need to remember.
POSTGRES_DB=attpcdaq             # The name of the database for PostgreSQL
DAQ_SECRET_KEY=[something]       # A secure, *STRONG* random string for Django's
↳ cryptography tools.
```

The user name for PostgreSQL is not important. Just set it to something reasonable. The remaining two things to be filled in are the PostgreSQL database password and the Django secret key. Set these both to **long**, random strings of characters since you will not need to remember them.

Warning: Although it may not seem that important to have a strong password on the local network, consider that the Django secret key is used to derive everything cryptography-related in the app. This means that it's especially important for this key to be both strong and secret.

One way to generate these random strings is the following Python script:

```
from __future__ import print_function
import random

chars = 'abcdefghijklmnopqrstuvwxyz0123456789!@#%^&*(_+=) '
sr = random.SystemRandom()
key = ''.join(sr.choice(chars) for i in range(50))
print(key)
```

Or, if you prefer a one-liner:

```
python -c "import random; chars = 'abcdefghijklmnopqrstuvwxyz0123456789!@#%^&*(_+=) ';"
↳ sr = random.SystemRandom(); print(''.join(sr.choice(chars) for i in range(50)))"
```

Building the containers

Once you've installed Docker and docker-compose, open a terminal in the root of the repository. This is the directory with the docker-compose.yml file. The Docker images can then be built with the command:

```
docker-compose build
```

This will create a set of Docker images and install all of the software’s dependencies inside them. This will require an internet connection.

Starting the app

Start all of the containers and the virtual network connecting them by running:

```
docker-compose up
```

This will instantiate the containers and start them, and then it will start printing the standard output from the containers. Keep this terminal window running to see the output as the program runs. If you want to quit the program later, press `Control-c` in this terminal.

The first time you run the code, it will need to do some housekeeping to get set up. This may take a minute or so. When the output printed to the terminal slows down or stops, continue with the next steps.

First-run setup

When the code is freshly installed, the database that backs the web app will be empty. We need to create a user in the web app so that we can log in and set up an experiment. To do this, open a new terminal and run this command:

```
docker exec -it attpcdaq_web_1 python manage.py createsuperuser
```

If we break this command down into parts, it opens a TTY inside the container running the Django app (`docker exec -it attpcdaq_web_1`) and runs the Django `manage.py` script to create a superuser account (`python manage.py createsuperuser`). It will prompt you for a username and password, which you should choose and remember for later.

Once you’ve made a superuser account, open a browser to <http://localhost:8080/admin> to access the Django admin interface. Log in with the username and password you just set up. This will put you on the Admin page.

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups	+ Add	Change
Users	+ Add	Change

DAQ

Config ids	+ Add	Change
Data routers	+ Add	Change
Data sources	+ Add	Change
Ecc servers	+ Add	Change
Experiments	+ Add	Change
Runs	+ Add	Change

LOGS

Log entries	+ Add	Change
-------------	-----------------------	------------------------

Recent actions

My actions

- [CoBo\[0\]](#)
Data source
- [+ DataRouter object](#)
Data router
- [+ ECCServer object](#)
Ecc server
- [x CoBo\[1\]](#)
Data source
- [x CoBo\[2\]](#)
Data source
- [+ RunMetadata object](#)
Run
- [+ Experiment object](#)
Experiment
- [test](#)
User
- [+ test](#)
User
- [+ RunMetadata object](#)
Run

This page allows you to access the internals of the DAQ web interface and directly change the contents of its database. For now, click on “Experiments” under the “DAQ” header and then click the “Add Experiment” button on the next page.

Click the green plus to add a new regular user account.

Note: Experiments are associated with user names in a one-to-one mapping in this program, so every time you add an experiment, you should also create a new experimental user to go along with it.

Also enter a name for the experiment. Data will be written into a directory with this name at the end of each run. Spaces are ok in this name. Finally, click “Save” to create the experiment.

Once you’ve finished this, click “Log Out” in the upper right to log out of the admin interface.

Starting the remote processes

Note: This section assumes the code is running on macOS. Linux distributions support a similar method of configuring a process to automatically launch using `systemd` services or `init` scripts, but that will not be covered here.

Under macOS, the remote GET processes are managed by `launchd`, the operating system’s main management process. It will automatically re-launch the processes if they fail, and it will coordinate logging of the processes’ standard outputs to a log file.

The behavior of `launchd` with respect to the GET software components is controlled by a Launch Agent plist file. Example plist files are included in the Git repository, but here is an annotated example for the ECC server:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
```

```
<plist version="1.0">
<dict>
  <!-- A label to identify the program -->
  <key>Label</key>
  <string>attpc.getEccSoapServer</string>

  <!-- Any necessary environment variables. This might include settings for paths
       needed for libraries installed using MacPorts, for example.-->
  <key>EnvironmentVariables</key>
  <dict>
    <key>DYLD_FALLBACK_LIBRARY_PATH</key>
    <string>/opt/local/lib</string>
  </dict>

  <!-- The commands needed to start the program. Each element of the command is
       ↪given in
       a separate <string> tag. The first element should be the full path to the
       ↪program,
       and the remaining elements give the command line arguments. -->
  <key>ProgramArguments</key>
  <array>
    <string>/path/to/getEccSoapServer</string>
    <string>--config-repo-url</string>
    <string>/path/to/configs/directory</string>
  </array>

  <!-- The working directory for the program. This is important for the dataRouter
       ↪as it's
       where that program will write the data. -->
  <key>WorkingDirectory</key>
  <string>/path/to/working/directory</string>

  <!-- Where to write the standard out and standard error files. These may be the
       ↪same file.
       It is probably best to put the logs in ~/Library/Logs since that will allow
       ↪you to
       view them with the Console application. -->
  <key>StandardOutPath</key>
  <string>/Users/USER/Library/Logs/getEccSoapServer.log</string>

  <key>StandardErrorPath</key>
  <string>/Users/USER/Library/Logs/getEccSoapServer.log</string>

  <!-- Keep the program running at all times, even if there are no incoming
       ↪connections. -->
  <key>KeepAlive</key>
  <true/>
</dict>
</plist>
```

A similar file should be created for the data router with the appropriate arguments.

Once plist files have been created, they are conventionally placed in `~/Library/LaunchAgents`, and they should be launched on startup if they are in that directory. To launch the programs manually, use `launchctl`:

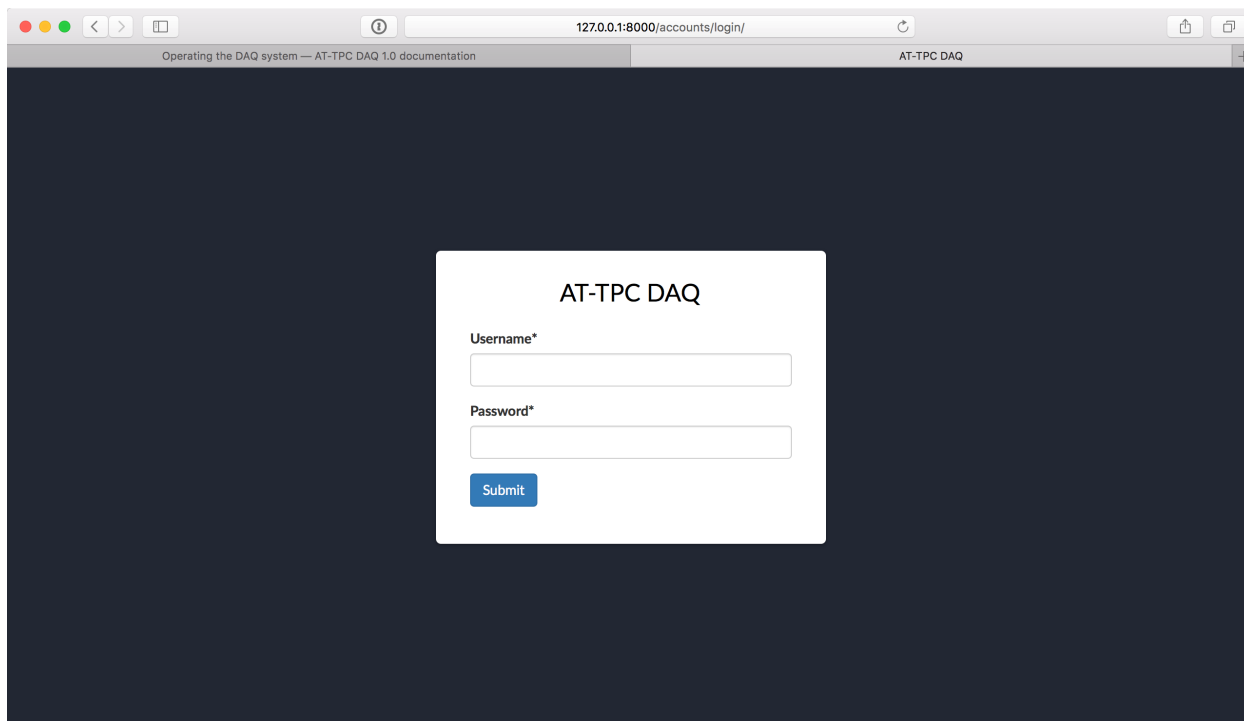
```
launchctl load ~/Library/LaunchAgents/attpc.getEccSoapServer.plist
```

Manually stopping the programs is very similar. Just replace `load` with `unload` in the above command.

This can also be automated for all of the remote computers using, e.g. Apple Remote Desktop.

Configuring the system

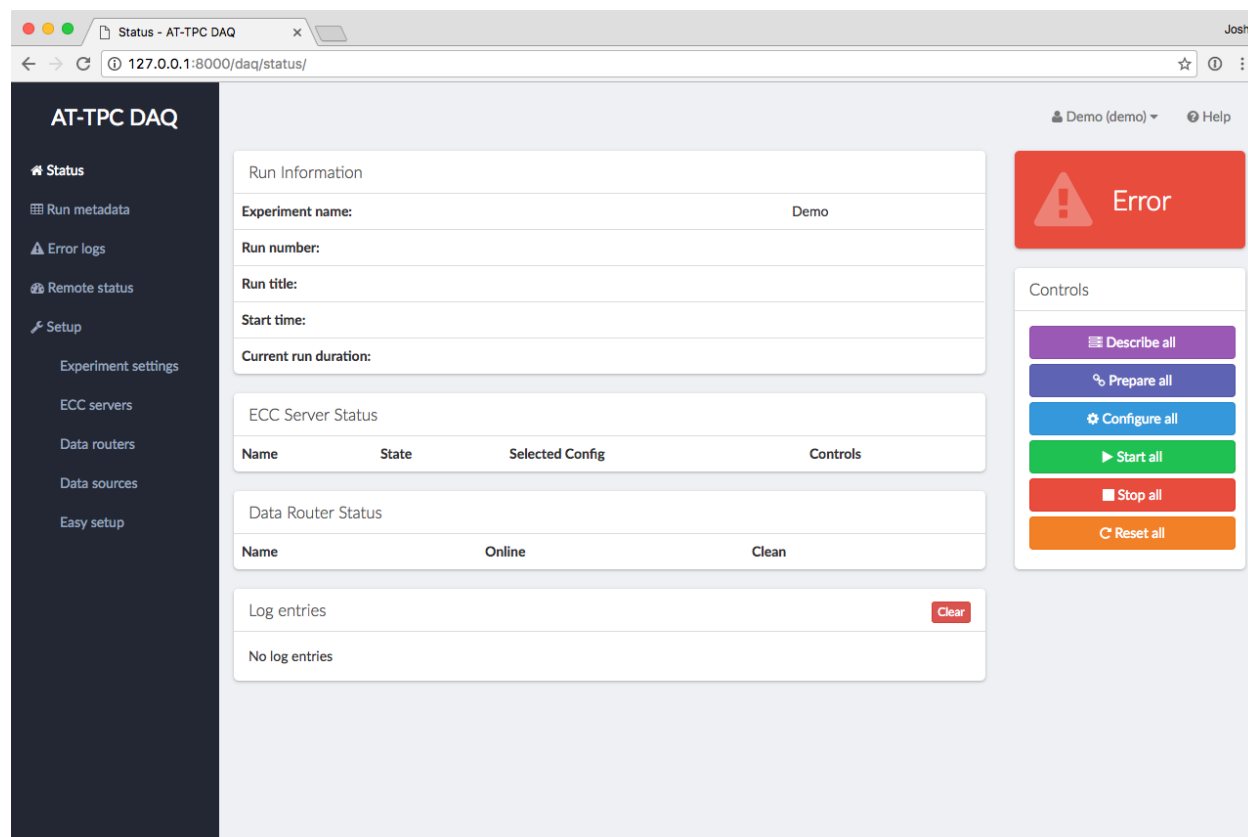
Once everything is up and running, the next step is to tell the DAQ system about the components you'll be using. First, log into the system. Go to <http://localhost:8080> in a browser to get to the main login page:



Sign in using the *experiment* account you created in the last section.

Note: Don't use the superuser account to log in here, or you'll just get an error page. That account should only be used to sign into the admin page.

After signing in, you'll find yourself on the main page. If you've just installed the system, it should be blank, like this:



We need to tell the system about the data routers and ECC servers in the system. That can be accomplished a few different ways, but the easiest method is to use the Easy Setup page.

Note: Just to be clear, all of the steps on this page are used to set up the *model* of the GET electronics in the DAQ GUI. These steps will not start `getEccSoapServer` or `dataRouter` processes. That must be done separately. This section just tells the system where to contact these processes, and we’re assuming that they’re already running and reachable from the network.

Easy setup

On the status page, click the “Easy setup” link in the left-hand menu column. This will take you to a form that you can fill out to automatically set up the system with some default values. Fill in values for the following fields:

Number of CoBos How many CoBos are you using? A data router will be created for each one.

Use one ECC server for all sources? If this is checked, the system will create one ECC server and link all CoBos to it. If this is unchecked, a separate ECC server object will be created for each CoBo.

IP address of first CoBo ECC server If we’re using one global ECC server, it will have this IP address. If each CoBo has its own ECC server, the first ECC server will get this IP address, and subsequent servers will get this address plus an offset in the last segment. For example, if this address is set to `192.168.1.10`, CoBo 0’s ECC server will be at `192.168.1.10`, CoBo 1’s ECC server will be at `192.168.1.11`, CoBo 2’s ECC server will be at `192.168.1.12`, etc.

IP address of first CoBo data router The address assigned to the data router of the first CoBo. Subsequent CoBos have an address that is incremented by an offset like above.

Is there a MuTAnT? If so, the system will create a data router object for it.

IP address of the MuTAnT ECC server If there is only one global ECC server, the MuTAnT will also be connected to that server, so this field will have no effect. Otherwise, the MuTAnT's ECC server will be found at this address.

IP address of MuTAnT data router The address where we should look for the MuTAnT data router.

Note: Again, just to be clear, the IP addresses entered here should be the addresses of the computers where the ECC server and data router processes are already running.

Once you click “Submit”, the system will create all of the necessary objects for this setup.

Danger: Submitting this form will overwrite the current DAQ GUI configuration. This will not destroy any data or config files, but it will remove any Data Router, ECC Server, and Data Source objects you've previously configured.

Manual configuration

If you need to tweak the results of the easy setup page, or if you need something more sophisticated than what it provides, you can always set things up manually. Under “Setup” in the left-hand navigation menu, there are links for setting up ECC servers, data routers, and data sources. Each of these leads to a table of the instances of that object that are currently set up. You can add a new instance using the “Add” button in the table header, and instances can be edited or removed using the buttons in each row.

To manually set up the system, you should first create your ECC servers and data routers. Then, create data source objects to link the two. For more information about the model used to describe the system, see *Modeling the system in code*.

Logging information about runs

In addition to controlling data taking, the DAQ system also allows you to record metadata about each run. This includes information about when the runs started and stopped along with metadata about the conditions during the run. This is intended to replace a physical log book with run data sheets. The set of items that are recorded is customizable, but there are a few fields which are always recorded.

Default run information

The default set of information will be recorded for every run, regardless of configuration. This set of fields includes the following:

- A sequential run number
- A run class identifying the type of the run. Options include “Testing”, “Production”, “Beam”, “Pulser”, and “Junk.”
- A title or label for the run
- The date and time when the run started and ended
- The name of the config file(s) used for this run

Adding additional fields

In addition to these defaults, any number of custom fields can be added. These fields, known in the DAQ software as *observables* can be used to record detector parameters like voltages and pressures. These should be set up at the beginning of an experiment, but they can also be added later.

To set up observables, click “Observables” under “Setup” in the left-hand menu. This will bring you to a list of the observables that are currently set up in the system. Add a new one by clicking the “Add” button in the top right corner of the “Observables” panel.

Tip: Observables in this list can be reordered by clicking and dragging the handle on the left-hand side of each row. This order will be remembered, and the fields for the observables will be presented in this order when entering run information.

An observable has four properties that you can set:

Name The name of the measurement. Choose something descriptive, but don’t include units. They will be added later.

Value type What type of data is this? Options include integer, floating point, and string values.

Units The units this will be recorded in. This is just for display, and **no unit conversions will be done** by the software.

Comment This optional comment will be shown next to the field on the run data sheet for this observable. This could be used to make a brief note of how to take a particular measurement, for example.

Fill these fields in and click “Submit” to add a new observable.

Operating the DAQ system

At this point, we’re nearly ready to take data. This page will describe how to choose a configuration file and start and stop runs. This is probably the most relevant part of the manual from the point of view of the person taking an experimental shift.

Web GUI status page

After logging into the system at <http://localhost:8080> or whatever address the system is available at, you will arrive at the main status page:

The screenshot shows the AT-TPC DAQ web interface. The left sidebar contains navigation links: Status, Run metadata, Measurements, Error logs, Setup, Experiment settings, Easy setup, ECC servers, Data routers, Data sources, and Observables. The main content area is divided into several panels:

- Run Information:** A form with fields for Experiment name (Demo), Run number, Run title, Run type, Start time, and Current run duration. It includes 'Update values' and 'Same as previous' buttons.
- ECC Server Status:** A table listing ECC servers with columns for Name, State, Logs, Selected Config, and Controls. The 'ECC' server is shown with a state of 'Idle' and a selected config of 'cobos-all/beam-all/beam-all'.
- Data Router Status:** A table listing data routers with columns for Name, Online, Clean, and Logs. All listed routers (DataRouter0 through DataRouter_mutant) show green checkmarks in the 'Online' and 'Clean' columns.
- Log entries:** A section showing 'No log entries' with a 'Clear' button.
- Controls:** A vertical stack of large buttons: 'Idle' (power icon), 'Describe all', 'Prepare all', 'Configure all', 'Start all', 'Stop all', and 'Reset all'.

This page shows an overview of what’s currently happening in the system. It is divided into a set of panels:

Run Information This panel has details about the current current run, like how long it has been going and what run number is currently being recorded.

ECC Server Status This panel lists the status of each ECC server the system knows about. The “State” indicator shows what state machine state the ECC server is in (i.e. “Idle”, “Ready”, “Running”, etc.). The “Selected Config” column lists the name of the config file set that will be used to configure the devices. The “Controls” column contains a set of buttons for changing the state of an individual ECC server. These button should only be used for troubleshooting purposes. Finally, clicking the icon in the “Logs” column will display the last few lines of the log file for that ECC server.

Data Router Status This panel shows the state of all of the data routers the system knows about. The “Online” column shows if the data router process is running, and the “Clean” column shows if the data router’s staging directory contains unsorted files. Both of these should display green checkmarks if the system is ready to take data.








Log Entries This panel will show the latest error messages from the web interface. This does not include error messages that may be produced by the GET software. You can click on an individual error to get more information and possibly a traceback. Finally, clicking “Clear” will discard all error messages.

Controls This set of large buttons configures the entire system at once. This is what you should use to control the system. The reset button will step the system back one state. For example, if the system is in the “Ready” state, pressing Reset will step it back to “Prepared”.

Selecting a configuration

Once all necessary processes are up and running, the ECC Server Status panel should display a status of “Idle” for each ECC server and the Data Router Status panel should show green check marks next to each data router. At this point, you should select a config file for each ECC server.

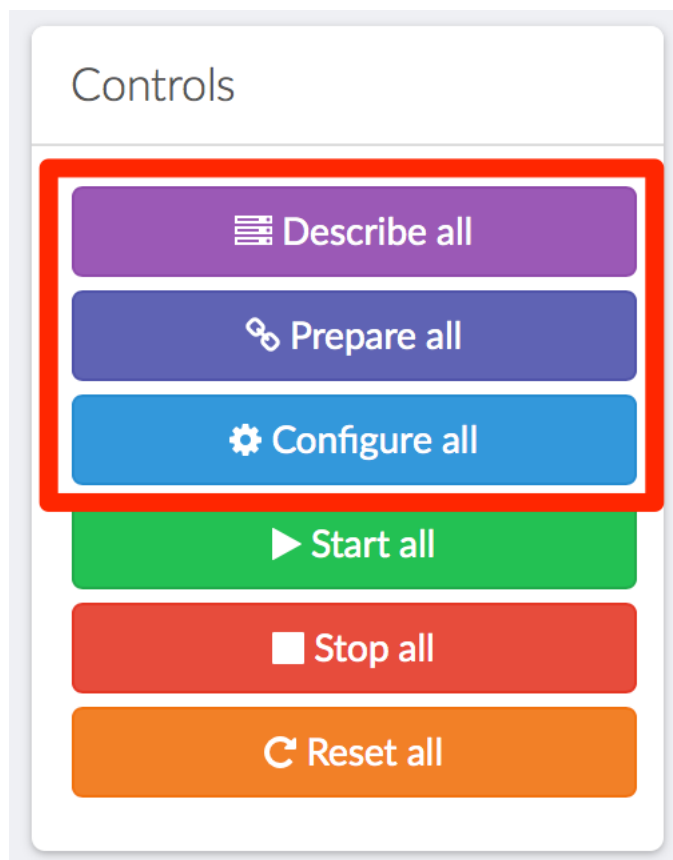
Config files can be selected by clicking the pencil icon next to the current config name in the Selected Config column of the ECC Server Status panel.

ECC Server Status				
Name	State	Logs	Selected Config	Controls
ECC	Idle	Q	cobos-all/beam-all/beam-all 	     

This will bring up a page with a drop-down menu listing the configurations available for that ECC server. The list of available configurations contains all possible permutations of the set of `describe-*.xcfg`, `prepare-*.xcfg`, and `configure-*.xcfg` files known to the ECC server. Each configuration is identified by a name composed of the names of the three `*.xcfg` files that go into it, formatted as `[describe-name]/[prepare-name]/[configure-name]`. For example, if you want to configure a data source using the files `describe-cobo0.xcfg`, `prepare-experiment.xcfg`, and `configure-experiment.xcfg`, then you should choose the configuration called `cobo0/experiment/experiment`. See [Config files](#) for more information about these files and their naming convention.

Preparing to take runs

After selecting a configuration, the CoBos and MuTAnT must be configured to prepare them to take data. This can be done using the first three buttons on the main Controls panel.



Begin by clicking the “Describe all” button. The system will then send a message to the ECC servers telling them to execute the “Describe” transition on the CoBos. The status label for each ECC server should then disappear and be replaced by a spinning cursor. Once the transition is finished, each ECC server should list a status of “Described”, and the overall system status in the top-right corner should also be shown as “Described.”

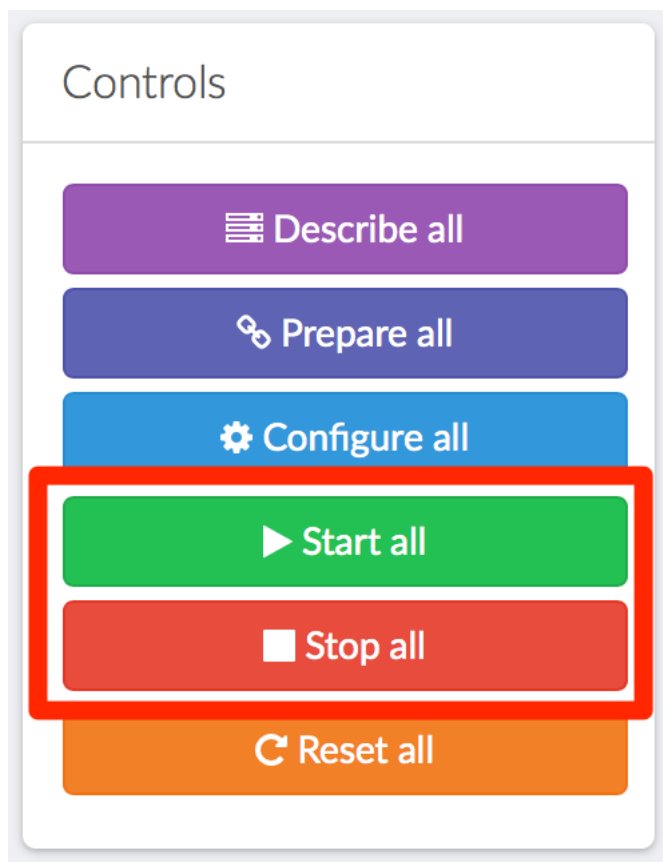
Note: These system-wide buttons only work if *all* ECC servers are in the same state. If they are in different states, you will need to use the individual controls in the ECC Server Status panel to bring them into the same state.

The next two steps are nearly identical. Click the “Prepare all” button, and wait until the status on each ECC server is shown as “Prepared.” Finally, click “Configure all,” and wait for a status of “Ready.” At this point, the system is ready to take data.

Note: If one or more of the CoBos fails to complete the state transition, their ECC servers will remain in whatever state they started in. This will be apparent since that ECC server will have a different label from the others, and the overall system status in the top-right corner will be shown as “Error.” If this happens, look for an error message in the “Log entries” panel at the bottom of the page, and try to diagnose the problem. Once the problem is fixed, try using the individual controls in the ECC Server Status panel to bring the troublesome server to the same state as the others.

Starting a run

Runs are controlled using the “Start all” and “Stop all” buttons in the main Controls panel.



Once you click “Start all,” the CoBos will begin recording data and the Run Information panel should update to reflect the new run.

Run Information		Update values	Same as previous
Experiment name:	Demo		
Run number:	0		
Run title:			
Run type:			
Start time:	Dec 12 2016, 20:25:34		
Current run duration:	00:00:37		

Danger: Data taking on the CoBos can also be started and stopped using the individual source control buttons on the ECC Server Status panel; however, if this is done, **the global run number will not be updated**. Therefore, these individual buttons should *only* be used in the case of an error where a CoBo fails to start recording data.

Recording run metadata

Once a new run has been started, metadata about the run can be entered by clicking on either the “Update values” or “Same as previous” button on the Run Information panel. Both of these will bring up a form where you can enter

information about the current run. The only difference between the two is that the “Same as previous” button will pre-fill some fields with their values from the previous run. This is useful for values that don’t change often.

Fill in any values on this page that were not filled automatically, and then click “Submit” to save them. You can get back to the status page by clicking “Status” in the left-hand menu.

Tip: The run will continue even if you navigate away from the status page or close the web browser.

Stopping a run

When it is time to stop a run, click the “Stop all” button. This will tell the CoBos to stop recording data, and it will also tell the system to connect to each computer where the data router is running and rearrange the data files into a directory for the just-completed run. Watch the “Clean” column in the Data Router Status panel to see when this process has finished.

Warning: It may take several seconds for the data files to be rearranged on each computer. You must wait until this process is complete before the system will allow you to start a new run.

Resetting the system

When an experiment is complete, or when you want to re-configure the CoBos, the system should be reset to the “Idle” state. This can be done using the “Reset all” button in the main Controls panel. One click of this button will step each ECC server back by one state in the state machine (see *CoBo state machine*).

Note: Each transition must finish before you click the Reset button again.

Developer documentation

This section contains some information about the structure of the DAQ code and some explanation of the design of the system. This will be useful mostly for people who want to modify the code or fix bugs.

Structure of the DAQ system

The AT-TPC DAQ system runs inside a collection of Docker containers. Each of these containers is responsible for running part of the system. In general, one container corresponds to one process. The responsibilities of each container are outlined below.

Django application

Container/service name: `web`

This is the core of the system, and it’s the container in which nearly all of the Python code in this application runs. The main process in this container is the Gunicorn web server, which runs the Django application that will be described in the next pages of this documentation.

This server responds to any requests for *dynamic* web content. When you click a link to load a page of the DAQ app, the Django library calls the appropriate functions in the web app to dynamically generate the HTML that will be shown. This also includes calls to the API that communicates with the ECC servers. These calls are implemented as functions that get called when certain URLs are requested.

NGINX web server

Container/service name: `nginx`

NGINX is a commonly used web server. It acts as a front-end to the application. When a URL is requested, NGINX receives the request first and decides whether the request is for static content or dynamic content. Requests for dynamically generated content are forwarded to the Gunicorn server described above for further processing. Requests for static content (such as CSS files, the help pages, and static images) are processed by NGINX itself in order to reduce the load on the Gunicorn server.

Celery task queue

Container/service name: `celery`

Celery is a Python-based, distributed, asynchronous task queue system. It receives messages from Django and schedules tasks accordingly. This allows asynchronous execution of portions of the web app’s code. For example, when you configure the CoBos, a set of tasks is sent to the Celery server that tell it to perform the configuration.

This is useful for long-running tasks like the configuration commands. If these tasks were executed synchronously inside the main Django process, the web interface would become unresponsive until the tasks finished. Instead, we execute the tasks asynchronously in the Celery worker processes and update the GUI later when the tasks are finished.

RabbitMQ message broker

Container/service name: `rabbitmq`

RabbitMQ is a “message broker” that coordinates communication between the main process of the Django application and the Celery task queue system. It needs to be running, but otherwise it is not particularly interesting from the perspective of the DAQ system.

PostgreSQL database

Container/service name: `db`

This is the database used to store the internal configuration of the web app. This stores things like the IP addresses of the ECC servers and data routers, the name of the config file to use for each CoBo, the history of recent runs, and the name of the current experiment.

Modeling the system in code

In the Django framework, *models* are used to represent entities. A model has a collection of *fields* associated with it, and these fields are mapped to columns in the model’s representation in the database. The models in the AT-TPC DAQ app are used to represent components of the DAQ system, including things like ECC servers and data routers. This page will provide an overview of the different models in the system and how they work together. For more specific information about each model, refer to their individual pages.

DAQ system components

The GET DAQ system is modeled using three classes: the *ECCServer*, the *DataRouter*, and the *DataSource*.

The ECC server

The *ECCServer* model is responsible for all communication with the GET ECC server processes. There should be one instance of this model for each ECC server in the system. The *ECCServer* has fields that store the IP address and port of the ECC server, and it also keeps track of which configuration file set to use, what the state of the ECC server is with respect to the CoBo state machine, and whether the ECC server is online and reachable.

In addition to storing basic information about the ECC servers, this model also has methods that allow it to communicate with the ECC server it represents. The *refresh_configs()* method fetches the list of available configuration file sets from the ECC server and stores it in the database. The *refresh_state()* method fetches the current CoBo state machine state from the ECC server and updates the *state* field accordingly. Finally, the method *change_state()* will tell the ECC server to transition its data sources to a different state. This last method is used to configure, start, and stop the CoBos during data taking.

Communication with the ECC server is done using the SOAP protocol. This is performed by a third-party library which is wrapped by the *EccClient* class in this module. The interface to the ECC server is defined by the file `web/attpcdaq/daq/ecc.wsdl`, which was copied from the source of the GET ECC server into this package. If the interface is updated in a future version of the ECC server, this file should be replaced.

The data router

The *DataRouter* model stores information about data routers in the system. The data router processes are each associated with one data source, and they record the data stream from that source to a GRAW file. This model simply stores information about the data router like its IP address, port, and connection type. This information is forwarded to the data sources when the ECC server configures them.

The data source

This represents a source of data, like a CoBo or a MuTAnT. This is functionally just a link between an ECC server, which controls the source, and a data router, which receives data from the source.

DAQ component models

<i>ECCServer</i> (*args, **kwargs)	Represents an individual ECC server which may control one or more data sources.
<i>DataRouter</i> (*args, **kwargs)	Represents the data router associated with one data source.
<i>DataSource</i> (*args, **kwargs)	A source of data, probably a CoBo or a MuTAnT.

attpcdag.dag.models.ECCServer

class attpcdag.dag.models.**ECCServer** (*args, **kwargs)

Represents an individual ECC server which may control one or more data sources.

This object is responsible for the bulk of the program's work. It is capable of communicating with the ECC server process to change the state of a CoBo or MuTAnT, and it also maintains a record of the ECC server's current state.

Data sources are associated with an ECC server through a many-to-one relationship, meaning that one ECC server may control many data sources. Alternatively, each data source may have its own ECC server, if that is desired.

Fields

name	A unique name for the ECC server
ip_address	The IP address of the ECC server
port	The TCP port that the ECC server listens on.
is_online	Whether the ECC server process is currently available and responding to requests
is_transitioning	Whether the ECC server is currently changing state
log_path	The path to the ECC server process's log file on the computer where the process is running.
selected_config	The configuration file set this ECC server will use
state	The state of the ECC server with respect to the CoBo state machine.

State constants and attributes

DESCRIBED	A constant representing the “described” state
IDLE	A constant representing the “idle” state
PREPARED	A constant representing the “prepared” state
READY	A constant representing the “ready” state
RUNNING	A constant representing the “running” state
RESET	A constant that is used to tell the system to step back-wards by one state
STATE_DICT	A dictionary mapping state constants back to state names

Methods

<code>change_state(target_state)</code>	Tells the ECC server to transition the data source to a new state.
<code>get_data_link_xml_from_clients()</code>	Get an XML representation of the data link for this source.
<code>refresh_configs()</code>	Fetches the list of configs from the ECC server and updates the database.
<code>refresh_state()</code>	Gets the current state of the data source from the ECC server and updates the database.
<code>_get_soap_client()</code>	Creates a SOAP client for communicating with the ECC server.
<code>_get_transition(client, current_state, ...)</code>	Look up the appropriate SOAP request to change the ECC server from one state to another.

attpcdag.daq.models.ECCServer.change_state

`ECCServer.change_state(target_state)`

Tells the ECC server to transition the data source to a new state.

If the request is successful, the `is_transitioning` field will be set to `True`, but the `state` field will *not* be updated automatically. To update this, `refresh_state()` should be called to see if the transition has completed.

Parameters `target_state (int)` – The desired final state. The required transition will be computed using `_get_transition()`.

Raises `RuntimeError` – If the data source does not have a config set.

attpcdag.daq.models.ECCServer.get_data_link_xml_from_clients

`ECCServer.get_data_link_xml_from_clients()`

Get an XML representation of the data link for this source.

This is used by the ECC server to establish a connection between the CoBo and the data router. The format is as follows:

```
<DataLinkSet>
  <DataLink>
    <DataSender id="[DataSource.name]">
    <DataRouter name="[DataSource.data_router_name]">
```

```
        ipAddress="[DataSource.data_router_ip_address]"
        port="[DataSource.data_router_port]"
        type="[DataSource.data_router_type]">
    </DataLink>
</DataLinkSet>
```

Returns The XML data.

Return type str

`attpcdaq.daq.models.ECCServer.refresh_configs`

`ECCServer.refresh_configs()`

Fetches the list of configs from the ECC server and updates the database.

If new configs are present on the ECC server, they will be added to the database. If configs are present in the database but are no longer known to the ECC server, they will be deleted.

The old configs are deleted based on their `last_fetched` field. Therefore, this field will be updated for each existing config set that is still present on the ECC server when this function is called.

`attpcdaq.daq.models.ECCServer.refresh_state`

`ECCServer.refresh_state()`

Gets the current state of the data source from the ECC server and updates the database.

This will update the `state` and `is_transitioning` fields of the `ECCServer`.

Raises `ECCError` – If the return code from the ECC server is nonzero.

`attpcdaq.daq.models.ECCServer._get_soap_client`

`ECCServer._get_soap_client()`

Creates a SOAP client for communicating with the ECC server.

The client loads the WSDL file, which describes the SOAP services, from the local disk. The target URL of the client is then set to the ECC server's address.

Returns The configured SOAP client.

Return type `EccClient`

`attpcdaq.daq.models.ECCServer._get_transition`

`classmethod ECCServer._get_transition(client, current_state, target_state)`

Look up the appropriate SOAP request to change the ECC server from one state to another.

Given the `current_state` and the `target_state`, this will either return the correct callable to make the transition, or it will raise an exception.

Parameters

- **client** (`EccClient`) – The SOAP client. One of its methods will be returned.
- **current_state** (`int`) – The current state of the ECC state machine.

- **target_state** (*int*) – The desired final state of the ECC state machine.

Returns The function corresponding to the requested transition. This can then be called with the appropriate arguments to change the ECC server’s state.

Return type function

Raises `ValueError` – If the requested states differ by more than one transition, or if no transition is needed.

attpcdag.daq.models.DataRouter

class `attpcdag.daq.models.DataRouter(*args, **kwargs)`

Represents the data router associated with one data source.

Each source of data (a CoBo or a MuTAnT) must be associated with a data router. The data router receives the data stream from the source and records it. This model stores information like the IP address, port, and type of the data router.

Fields

<code>name</code>	A unique name for the data router
<code>ip_address</code>	The IP address of the data router
<code>port</code>	The TCP port where the data router is listening.
<code>connection_type</code>	The protocol of the data router.
<code>is_online</code>	Whether the data router is online and available
<code>log_path</code>	The path to the log file on the computer where the data router is running.
<code>staging_directory_is_clean</code>	Whether the directory where the data router is running contains any GRAW files.

Data router type constants

<code>FDT</code>	A constant for the “FDT” protocol.
<code>ICE</code>	A constant for the “ICE” protocol
<code>TCP</code>	A constant for the “TCP” protocol.
<code>ZBUF</code>	A constant for the “ZBUF” protocol

attpcdag.daq.models.DataSource

class `attpcdag.daq.models.DataSource(*args, **kwargs)`

A source of data, probably a CoBo or a MuTAnT.

This model represents a source of data in the system, like a CoBo or a MuTAnT. A data source is controlled by an ECC server, and it sends its data to a data router. Therefore, this is simply a link between an *ECCServer* instance and a *DataRouter* instance.

Fields

name	A unique name for the data source.
ecc_server	The <i>ECCServer</i> that controls this data source.
data_router	The <i>DataRouter</i> that receives the data stream from this source.

Methods

<code>get_data_link_xml()</code>	Get an XML representation of the data link for this source.
----------------------------------	---

`attpcdaq.daq.models.DataSource.get_data_link_xml`

`DataSource.get_data_link_xml()`

Get an XML representation of the data link for this source.

This is used by the ECC server to establish a connection between the CoBo or MuTAnT and the data router. The format is as follows:

```
<DataLink>
  <DataSender id="[DataSource.name]">
  <DataRouter name="[DataSource.data_router_name]"
    ipAddress="[DataSource.data_router_ip_address]"
    port="[DataSource.data_router_port]"
    type="[DataSource.data_router_type]">
</DataLink>
```

This must be wrapped in `<DataLinkSet>` tags before sending it to the ECC server.

Returns The XML data for this source.

Return type `xml.etree.ElementTree.Element`

Config file sets

Sets of config files are represented as *ConfigId* objects. These contain fields for each of the three config files for the three configuration steps. These sets will generally be created automatically by fetching them from the ECC servers using `ECCServer.refresh_configs()`, but they can also be created manually if necessary.

Config file models

<code>ConfigId(*args, **kwargs)</code>	Represents a configuration file set as seen by the ECC servers.
--	---

`attpcdaq.daq.models.ConfigId`

class `attpcdaq.daq.models.ConfigId(*args, **kwargs)`

Represents a configuration file set as seen by the ECC servers.

This will generally be retrieved from the ECC servers using a SOAP call. If this is the case, an object can be constructed from the XML representation using the class method `from_xml`.

It is important to note that this is just a representation of the config files which is used for communicating with the ECC server. No actual configuration is done by this program.

Note: This model stores configuration names using the convention of the ECC server. This means that the actual filenames seen by the ECC server will be, for example, `describe-[name].xcfg`. The prefix and file extension are added automatically by the ECC server.

Fields

<code>describe</code>	The name of the configuration for the “describe” step
<code>prepare</code>	The name of the configuration for the “prepare” step
<code>configure</code>	The name of the configuration for the “configure” step
<code>ecc_server</code>	The ECC server that this configuration set is associated with
<code>last_fetched</code>	The date and time when this config was fetched from the ECC server.

Methods

<code>as_xml()</code>	Get an XML representation of the object.
<code>from_xml(node)</code>	Construct a ConfigId object from the given XML representation.

attpcdag.daq.models.ConfigId.as_xml

`ConfigId.as_xml()`

Get an XML representation of the object.

This is useful for sending to the ECC server. The format is as follows:

```
<ConfigId>
  <SubConfigId type="describe">[self.describe]</SubConfigId>
  <SubConfigId type="prepare">[self.prepare]</SubConfigId>
  <SubConfigId type="configure">[self.configure]</SubConfigId>
</ConfigId>
```

Returns The XML representation.

Return type str

attpcdag.daq.models.ConfigId.from_xml

classmethod `ConfigId.from_xml (node)`

Construct a ConfigId object from the given XML representation.

Parameters `node` (`xml.etree.ElementTree.Element` or `str`) – The XML representation of the object, probably from the ECC server. If it’s a string, it will be automatically converted to the appropriate XML node object.

Returns `new_config` – A `ConfigId` object constructed from the representation. Note that this object is **not** automatically committed to the database, so one should call `new_config.save()` if that is desired.

Return type *ConfigId*

Run and experiment metadata

The *Experiment* and *RunMetadata* models store information about the experiment and the runs it contains. They are used to number the runs and to store metadata like the experiment name, the duration of each run, and a comment describing the conditions for each run.

The *Observable* and *Measurement* classes are used to store measurements of experimental parameters like voltages, pressures, and scalars. An *Observable* defines a quantity that can be measured, and each one adds a new field that can be filled in on the Run Info sheet. When a user fills in values for an *Observable*, a corresponding *Measurement* object is created to store that value. This design was chosen so that the user can add new observables at any time without reloading the code or altering the database structure. This would not be possible if we just defined a new field on the *RunMetadata* object for each observable.

Metadata models

<i>Experiment</i> (*args, **kwargs)	Represents an experiment and the settings relevant to one.
<i>RunMetadata</i> (*args, **kwargs)	Represents the metadata describing a data run.
<i>Observable</i> (*args, **kwargs)	Something that can be measured.
<i>Measurement</i> (*args, **kwargs)	A measurement of an Observable.

attpcdaq.daq.models.Experiment

class attpcdaq.daq.models.**Experiment** (*args, **kwargs)

Represents an experiment and the settings relevant to one.

This model keeps track of run numbers and knows the name of the experiment. It is queried when rearranging data files at the end of a run, when the experiment name is used as the name of the directory in which to store the files.

Fields

name	The name of the experiment.
is_active	Is this the active experiment? Only one experiment may be active at a time.

Properties

<i>is_running</i>	Whether a run is currently being recorded.
<i>latest_run</i>	Get the most recent run in the experiment.
<i>next_run_number</i>	Get the number that the next run should have.

attpcdag.daq.models.Experiment.is_running`Experiment.is_running`

Whether a run is currently being recorded.

Returns True if the latest run has started but not stopped. False otherwise (including if there are no runs).

Return type bool

attpcdag.daq.models.Experiment.latest_run`Experiment.latest_run`

Get the most recent run in the experiment.

This will return the current run if a run is ongoing, or the most recent run if the DAQ is stopped.

Returns The most recent or current run. If there are no runs for this experiment, None will be returned instead.

Return type RunMetadata or None

attpcdag.daq.models.Experiment.next_run_number`Experiment.next_run_number`

Get the number that the next run should have.

The number returned is the run number from `latest_run` plus 1. Therefore, if a run is currently being recorded, this function will return the current run number plus 1.

If there are no runs, this will return 0.

Returns The next run number.

Return type int

Methods

<code>start_run()</code>	Creates and saves a new <i>RunMetadata</i> object with the next run number for the experiment.
<code>stop_run()</code>	Stops the current run.
<code>save(*args, **kwargs)</code>	Override of save to enforce only one active experiment at a time.

attpcdag.daq.models.Experiment.start_run`Experiment.start_run()`

Creates and saves a new *RunMetadata* object with the next run number for the experiment.

The `start_datetime` field of the created *RunMetadata* instance is set to the current date and time.

Raises `RuntimeError` – If there is already a run that has started but not stopped.

attpcdag.daq.models.Experiment.stop_run

`Experiment.stop_run()`

Stops the current run.

This sets the `stop_datetime` of the current run to the current date and time, effectively ending the run.

Raises `RuntimeError` – If there is no current run.

attpcdag.daq.models.Experiment.save

`Experiment.save(*args, **kwargs)`

Override of save to enforce only one active experiment at a time.

attpcdag.daq.models.RunMetadata

class `attpcdag.daq.models.RunMetadata(*args, **kwargs)`

Represents the metadata describing a data run.

Fields can be added to this model to store any type of data we want to record about each run. For instance, a title can be added so we know what the run was recording.

Fields

<code>experiment</code>	The experiment that this run is a part of
<code>run_number</code>	The run number
<code>start_datetime</code>	The date and time when the run started
<code>stop_datetime</code>	The date and time when the run ended
<code>title</code>	A title or comment describing the run

Properties

<code>duration</code>	Get the duration of the run.
<code>duration_string</code>	Get the duration as a string.

attpcdag.daq.models.RunMetadata.duration

`RunMetadata.duration`

Get the duration of the run.

If the run has not ended, the difference is taken with respect to the current time.

Returns Object representing the duration of the run.

Return type `datetime.timedelta`

attpcdag.dag.models.RunMetadata.duration_string`RunMetadata.duration_string`

Get the duration as a string.

Returns The duration of the current run. The format is HH:MM:SS.**Return type** str**attpcdag.dag.models.Observable****class** attpcdag.dag.models.**Observable** (*args, **kwargs)

Something that can be measured.

Observables correspond to columns in a run sheet. Add a new one to add a new field to the run sheet.

Fields

name	The name of the observable
experiment	The experiment that this observable is associated with.
comment	A comment to describe how to take a measurement, for example.
units	The units that measurements will be recorded in.
value_type	The data type of the measurement.

Value type constants

FLOAT	Constant for a floating-point measurement
INTEGER	Constant for an integer measurement
STRING	Constant for a string measurement

attpcdag.dag.models.Measurement**class** attpcdag.dag.models.**Measurement** (*args, **kwargs)

A measurement of an Observable.

Measurements are like instances of Observables. When you fill in the run sheet, a Measurement is created for each Observable-related field on the sheet.

Fields

observable	The Observable that this is a measurement of
run_metadata	The run that this measurement is for
serialized_value	The value as a string

Other attributes and properties

<code>python_type</code>	The Python data type we expect for this measurement.
<code>value</code>	The value, converted to the expected data type.

Interacting with the system

Interaction with the Django web app occurs through *views*, which are just functions and classes that Django calls when certain URLs are requested. Views are used to render the pages of the web app, and they are also how the user tells the system to “do something” like configure a CoBo or refresh the state of an ECC server.

Views are mapped to URLs automatically by Django. This mapping is set up in the module `attpcdaq.daq.urls`.

Some views render pages that accept information from the user. These generally use a Django form class to process the data.

Since the views serve a number of different purposes, they are organized into a few separate modules in the package `attpcdaq.daq.views`.

Page rendering views

These views, located in the module `attpcdaq.daq.views.pages`, are used to render the pages of the web app. This includes functions like `status()`, which renders the main status page, and others like `show_log_page()`, which contacts a remote computer, fetches the end of a log file, and renders a page showing it.

Views

<code>status(request)</code>	Renders the main status page.
<code>choose_config(request, pk)</code>	Renders a page for choosing the config for an ECC server.
<code>experiment_settings(request)</code>	Renders the experiment settings page.
<code>show_log_page(request, pk, program)</code>	Retrieve and render the log file for the given program.
<code>EasySetupPage(**kwargs)</code>	Renders the easy setup page, where the system can be set up in one step.

`attpcdaq.daq.views.pages.status`

`attpcdaq.daq.views.pages.status(request)`

Renders the main status page.

Parameters `request` (*HttpRequest*) – The request object.

Returns The rendered page.

Return type `HttpResponse`

`attpcdaq.daq.views.pages.choose_config`

`attpcdaq.daq.views.pages.choose_config(request, pk)`

Renders a page for choosing the config for an ECC server.

This renders the `ConfigSelectionForm` to pick the configuration.

Parameters

- **request** (*HttpRequest*) – The request object
- **pk** (*int*) – The primary key of the ECC server to configure.

Returns Redirects back to the main page on success.

Return type `HttpResponse`

`attpcdag.daq.views.pages.experiment_settings`

`attpcdag.daq.views.pages.experiment_settings(request)`

Renders the experiment settings page.

`attpcdag.daq.views.pages.show_log_page`

`attpcdag.daq.views.pages.show_log_page(request, pk, program)`

Retrieve and render the log file for the given program.

This can be used to display the end of the log file for the ECC server process or the data router process.

Parameters

- **request** (*HttpRequest*) – The request object.
- **pk** (*int*) – The integer primary key of the data source whose logs we want to view.
- **program** (*str*) – The program whose logs we want. Must be one of 'ecc' or 'data_router'.

Returns Renders the `log_file.html` template with the given log file as content.

Return type `HttpResponse`

`attpcdag.daq.views.pages.EasySetupPage`

`class attpcdag.daq.views.pages.EasySetupPage(**kwargs)`

Renders the easy setup page, where the system can be set up in one step.

Methods

`form_valid(form)`

Checks that the form data is valid, and then performs the setup.

Backend functions

`easy_setup(experiment, num_cobos, ..., ...)`

Create a set of model instances with default values based on the given parameters.

attpcdaq.daq.views.pages.easy_setup

```
attpcdaq.daq.views.pages.easy_setup(experiment, num_cobos, one_ecc_server,  
                                     first_cobo_ecc_ip, first_cobo_data_router_ip,  
                                     cobo_ecc_log_path, cobo_router_log_path,  
                                     cobo_config_root, cobo_config_backup_root,  
                                     mutant_is_present=False, mutant_ecc_ip=None,  
                                     mutant_data_router_ip=None, mu-  
                                     tant_ecc_log_path=None, mu-  
                                     tant_router_log_path=None, mutant_config_root=None,  
                                     mutant_config_backup_root=None)
```

Create a set of model instances with default values based on the given parameters.

This will populate the database with all of the required DAQ components. Note that all old instances will be deleted. This is done atomically, so if this function fails, nothing will be changed.

Parameters

- **experiment** (`attpcdaq.daq.models.Experiment`) – The experiment to modify.
- **num_cobos** (`int`) – The number of CoBos to add to the system.
- **one_ecc_server** (`bool`) – If True, all data sources will use the same ECC server. If False, a separate ECC server will be created for each data source.
- **first_cobo_ecc_ip** (`str`) – The IP address of the ECC server for the first CoBo. Subsequent ECC servers will have IP addresses whose last component is incremented by one.
- **first_cobo_data_router_ip** (`str`) – The IP address of the data router for the first CoBo. Subsequent data routers will have IP addresses whose last component is incremented by one.
- **cobo_ecc_log_path** (`str`) – The path to the CoBo ECC server log file. This is on the remote computer.
- **cobo_router_log_path** (`str`) – The path to the CoBo data router log file. This is on the remote computer.
- **cobo_config_root** (`str`) – The path to the config file directory on the remote computer.
- **cobo_config_backup_root** (`str`) – The path to which config files should be backed up on the remote computer.
- **mutant_is_present** (`bool`, *optional*) – True if the MuTAnT is present in the system and should be set up.
- **mutant_ecc_ip** (`str`, *optional*) – The IP address of the ECC server of the MuTAnT. This will be overridden if *one_ecc_server* is True.
- **mutant_data_router_ip** (`str`, *optional*) – The IP address of the data router of the MuTAnT.
- **mutant_ecc_log_path** (`str`, *optional*) – The path to the MuTAnT ECC server log file. This is on the remote computer.
- **mutant_router_log_path** (`str`, *optional*) – The path to the MuTAnT data router log file. This is on the remote computer.
- **mutant_config_root** (`str`, *optional*) – The path to the config file directory on the remote computer.

- **mutant_config_backup_root** (*str*, *optional*) – The path to which config files should be backed up on the remote computer.

ECC interaction views

A few of the views in the module `attpcdaq.daq.views.api` are used to interact with the ECC servers and request that they perform some action. These views are called when the user clicks a button to request a state change.

<code>source_change_state(request)</code>	Submits a request to tell the ECC server to change a source's state.
<code>source_change_state_all(request)</code>	Send requests to change the state of all ECC servers.

attpcdaq.daq.views.api.source_change_state

`attpcdaq.daq.views.api.source_change_state(request)`

Submits a request to tell the ECC server to change a source's state.

The transition request is put in the Celery task queue.

Parameters **request** (*HttpRequest*) – The request must include the primary key `pk` of the ECC server and the integer `target_state` to change to. The request must be made via POST.

Returns The JSON response includes the items outlined in `_make_status_response`.

Return type `JsonResponse`

attpcdaq.daq.views.api.source_change_state_all

`attpcdaq.daq.views.api.source_change_state_all(request)`

Send requests to change the state of all ECC servers.

The requests are queued to be performed asynchronously.

Parameters **request** (*HttpRequest*) – The request method must be POST, and it must contain an integer representing the target state.

Returns A JSON array containing status information about all ECC servers.

Return type `JsonResponse`

API views

The remaining views in the `attpcdaq.daq.views.api` module provide an interface to the information stored in the database. These generate pages that allow the user to add, modify, and remove instances of models. There are also views that return information from the database so the GUI can be updated by AJAX calls.

Unlike other views described above, the API views for manipulating database objects are based on classes instead of functions. These are all subclasses of generic views provided by Django, so for more information on these views, take a look at Django's documentation for class-based views.

Refreshing data

`refresh_state_all(request)`Fetch the state of all data sources from the database and return the overall state of the system.

`attpcdaq.daq.views.api.refresh_state_all`

`attpcdaq.daq.views.api.refresh_state_all(request)`

Fetch the state of all data sources from the database and return the overall state of the system.

The value of the data source state that will be returned is whatever the database says. These values will be returned along with the overall state of the system and some information about the current experiment and run.

Note: This function does *not* communicate with the ECC server in any way. To contact the ECC server and update the state stored in the database, call `attpcdaq.daq.models.ECCServer.refresh_state()` instead.

The JSON array returned will contain the following keys:

overall_state The overall state of the system. If all of the data sources have the same state, this should be the numerical ID of a state. If the sources have different states, it should be -1.

overall_state_name The name of the overall state of the system. Either a state name or “Mixed” if the state is inconsistent.

run_number The current run number.

start_time The date and time when the current run started.

run_duration The duration of the current run. This is with respect to the current time if the run has not ended.

individual_results The results for the individual data sources. These are sub-arrays.

The sub arrays for the individual results should include the keys:

success Whether the request succeeded.

pk The primary key of the source.

error_message An error message.

state The ID of the current state.

state_name The name of the current state

transitioning Whether the source is undergoing a state transition.

Parameters **request** (*HttpRequest*) – The request object. The method must be GET.

Returns An array of dictionaries containing the results from each data source. See above for the contents.

Return type JsonResponse

Working with data sources

`AddDataSourceView(**kwargs)`

Add a data source.

`ListDataSourcesView(**kwargs)`

List all data sources.

Continued on next page

Table 1.27 – continued from previous page

<code>UpdateDataSourceView(**kwargs)</code>	Change parameters on a data source.
<code>RemoveDataSourceView(**kwargs)</code>	Delete a data source.

Working with data routers

<code>AddDataRouterView(**kwargs)</code>	Add a data router.
<code>ListDataRoutersView(**kwargs)</code>	List all data routers.
<code>UpdateDataRouterView(**kwargs)</code>	Modify a data router.
<code>RemoveDataRouterView(**kwargs)</code>	Delete a data router.

Working with ECC servers

<code>AddECCServerView(**kwargs)</code>	Add an ECC server.
<code>ListECCServersView(**kwargs)</code>	List all ECC servers.
<code>UpdateECCServerView(**kwargs)</code>	Modify an ECC server.
<code>RemoveECCServerView(**kwargs)</code>	Delete an ECC server.

Working with run metadata

<code>ListRunMetadataView(**kwargs)</code>	List the run information for all runs.
<code>UpdateRunMetadataView(**kwargs)</code>	Change run metadata
<code>UpdateLatestRunMetadataView(**kwargs)</code>	Redirects to <code>UpdateRunMetadataView</code> for the latest run.

Working with Observables

<code>AddObservableView(**kwargs)</code>	Add a new observable to the experiment.
<code>ListObservablesView(**kwargs)</code>	List the observables registered for this experiment.
<code>UpdateObservableView(**kwargs)</code>	Change properties of an Observable.
<code>RemoveObservableView(**kwargs)</code>	Remove an observable from this experiment.

Setting the ordering of observables

<code>set_observable_ordering(request)</code>	An AJAX request that sets the order in which observables are displayed.
---	---

attpcdaq.daq.views.api.set_observable_ordering

`attpcdaq.daq.views.api.set_observable_ordering(request)`

An AJAX request that sets the order in which observables are displayed.

The request should be submitted via POST, and the request body should be JSON encoded. The content should be a dictionary with the key “new_order” mapped to a list of Observable primary keys in the desired order.

Parameters **request** (*HttpRequest*) – The request with the information given above. Must be POST.

Returns If successful, the JSON data `{ 'success' : True }` is returned.

Return type `JsonResponse`

Helper functions

These helper functions are called by some of the views to avoid duplicating code. They are located in the module `attpcdaq.daq.views.helpers`.

<code>calculate_overall_state(request)</code>	Find the overall state of the system.
<code>get_ecc_server_statuses(request)</code>	Gets some information about the ECC servers.
<code>get_data_router_statuses(request)</code>	Gets some information about the data routers.
<code>get_status(request)</code>	Returns some information about the system's status.

`attpcdaq.daq.views.helpers.calculate_overall_state`

`attpcdaq.daq.views.helpers.calculate_overall_state(request)`

Find the overall state of the system.

Parameters **request** (*django.http.request.HttpRequest*) – The request object.

Returns

- **overall_state** (*int or None*) – The overall state of the system. Returns `None` if the state is mixed.
- **overall_state_name** (*str*) – The name of the system state. The value 'Mixed' is returned if the system is not in a consistent state.

`attpcdaq.daq.views.helpers.get_ecc_server_statuses`

`attpcdaq.daq.views.helpers.get_ecc_server_statuses(request)`

Gets some information about the ECC servers.

This produces a dictionary with the following key-value pairs:

'success' Whether the request succeeded.

'error_message' An error message, if applicable.

'pk' The integer primary key of the ECC server in the database.

'state' The current (integer) state of the ECC server, as enumerated in the constants attached to that class.

'state_name' The name of the current state of the ECC server.

'transitioning' Whether the ECC server is transitioning between states.

Returns A dictionary with the above keys.

Return type `dict`

attpcdag.daq.views.helpers.get_data_router_statuses

`attpcdag.daq.views.helpers.get_data_router_statuses(request)`

Gets some information about the data routers.

This produces a dictionary with the following key-value pairs:

‘**success**’ Whether the request succeeded.

‘**pk**’ The integer primary key of the data router in the database.

‘**is_online**’ Whether the router is available.

‘**is_clean**’ Whether the staging directory is clean.

Returns A dictionary of the values above.

Return type dict

attpcdag.daq.views.helpers.get_status

`attpcdag.daq.views.helpers.get_status(request)`

Returns some information about the system’s status.

This generates a dictionary containing the following key-value pairs:

‘**overall_state**’ The overall state of the system. If all of the data sources have the same state, this should be the numerical ID of a state. If the sources have different states, it should be -1.

‘**overall_state_name**’ The name of the overall state of the system. Either a state name or “Mixed” if the state is inconsistent.

‘**run_number**’ The current run number.

‘**run_title**’ The title of the current run.

‘**run_class**’ The type of the current run.

‘**start_time**’ The date and time when the current run started.

‘**run_duration**’ The duration of the current run. This is with respect to the current time if the run has not ended.

‘**ecc_server_status_list**’ Status of each ECC server. See `get_ecc_server_statuses()` for details.

‘**data_router_status_list**’ Status of each data router. See `get_data_router_statuses()` for details.

This is helpful when generating JSON responses to update the main page periodically.

Parameters **request** (*HttpRequest*) – The request object. This must be included so we can get the name of the current user when fetching the *Experiment* object.

Returns A dictionary containing the information above.

Return type dict

Interfacing with the remote processes

The `attpcdag.daq.workertasks` module contains a class that uses the Paramiko SSH library to connect to the nodes running the data router and ECC server in order to, for example, organize files at the end of a run. It can also check whether these processes are running.

This class should typically be used as a context manager (i.e., with a `with` statement). For example, to organize files, you could try the following:

```
with WorkerInterface(data_router_ip_address) as wint:
    wint.organize_files(experiment_name, run_number)
```

When used in this manner, the SSH session will automatically be opened when entering the `with` block and closed when leaving it.

The WorkerInterface class

class `attpcdaq.daq.workertasks.WorkerInterface` (*hostname*, *port=22*, *username=None*, *config_path=None*)

An interface to perform tasks on the DAQ worker nodes.

This is used to perform tasks on the computers running the data router and the ECC server. This includes things like cleaning up the data files at the end of each run.

The connection is made using SSH, and the SSH config file at `config_path` is honored in making the connection. Additionally, the server *must* accept connections authenticated using a public key, and this public key must be available in your `.ssh` directory.

Parameters

- **hostname** (*str*) – The hostname to connect to.
- **port** (*int*, *optional*) – The port that the SSH server is listening on. The default is 22.
- **username** (*str*, *optional*) – The username to use. If it isn't provided, a username will be read from the SSH config file. If no username is listed there, the name of the user running the code will be used.
- **config_path** (*str*, *optional*) – The path to the SSH config file. The default is `~/.ssh/config`.

Methods

<code>find_data_router()</code>	Find the working directory of the data router process.
<code>get_graw_list()</code>	Get a list of GRAW files in the data router's working directory.
<code>working_dir_is_clean()</code>	Check if there are GRAW files in the data router's working directory.
<code>check_ecc_server_status()</code>	Checks if the ECC server is running.
<code>check_data_router_status()</code>	Checks if the data router is running.
<code>organize_files(experiment_name, run_number)</code>	Organize the GRAW files at the end of a run.
<code>tail_file(path[, num_lines])</code>	Retrieve the tail of a text file on the remote host.

`attpcdaq.daq.workertasks.WorkerInterface.find_data_router`

`WorkerInterface.find_data_router()`

Find the working directory of the data router process.

The directory is found using `ls -of`, which must be available on the remote system.

Returns The directory where the data router is running, and therefore writing data.

Return type str

Raises RuntimeError – If lsof finds something strange instead of a process called dataRouter.

attpcdaq.daq.workertasks.WorkerInterface.get_graw_list

WorkerInterface.get_graw_list()

Get a list of GRAW files in the data router's working directory.

Returns A list of the full paths to the GRAW files.

Return type list[str]

attpcdaq.daq.workertasks.WorkerInterface.working_dir_is_clean

WorkerInterface.working_dir_is_clean()

Check if there are GRAW files in the data router's working directory.

Returns True if there are files in the working directory, False otherwise.

Return type bool

attpcdaq.daq.workertasks.WorkerInterface.check_ecc_server_status

WorkerInterface.check_ecc_server_status()

Checks if the ECC server is running.

Returns True if getEccSoapServer is running.

Return type bool

attpcdaq.daq.workertasks.WorkerInterface.check_data_router_status

WorkerInterface.check_data_router_status()

Checks if the data router is running.

Returns True if dataRouter is running.

Return type bool

attpcdaq.daq.workertasks.WorkerInterface.organize_files

WorkerInterface.organize_files(experiment_name, run_number)

Organize the GRAW files at the end of a run.

This will get a list of the files written in the working directory of the data router and move them to the directory ./experiment_name/run_name, which will be created if necessary. For example, if the experiment_name is "test" and the run_number is 4, the files will be placed in ./test/run_0004.

Parameters

- **experiment_name** (str) – A name for the experiment directory.

- **run_number** (*int*) – The current run number.

attpcdag.daq.workertasks.WorkerInterface.tail_file

WorkerInterface.**tail_file** (*path*, *num_lines*=50)

Retrieve the tail of a text file on the remote host.

Note that this assumes the file is ASCII-encoded plain text.

Parameters

- **path** (*str*) – Path to the file.
- **num_lines** (*int*) – The number of lines to include.

Returns The tail of the file’s contents.

Return type `str`

Asynchronous tasks and Celery

Due to the distributed design of the DAQ system, it’s very likely that sometimes a command sent to the system will take a while to process. This is especially true when communicating with an ECC server if the ECC server is configuring all of its attached data sources in series. If we decided to send a long-running command to the ECC server synchronously in the middle of whatever view was responding to the user’s HTTP request, the view would block on the communication until it finished. This would prevent it from updating the GUI, giving the impression that the software has crashed, and in extreme cases, the browser could even return a timeout error.

To prevent this problem, we process slow commands *asynchronously* with Celery. Instead of directly initiating communications, the view submits a task to the Celery queue and returns immediately, updating the GUI to indicate that the task is processing. When the task is completed, some part of the database is generally updated. The GUI is then updated to reflect the fact that the task has completed when it periodically refreshes itself.

Tasks

The Celery tasks in this application are just Python functions with the `@shared_task` decorator. This decorator registers them with the Celery system as tasks, and it also allows us to set a time limit on them. All of the tasks are located in the module `attpcdag.daq.tasks`.

ECC server interaction

<code>eccserver_refresh_state_task</code>	Fetch the state of the given ECC server.
<code>eccserver_refresh_all_task</code>	Fetch the state of all ECC servers.
<code>eccserver_change_state_task</code>	Change the state of an ECC server (make it perform a transition).

attpcdag.daq.tasks.eccserver_refresh_state_task

```
attpcdag.daq.tasks.eccserver_refresh_state_task = <@task: attpcdag.daq.tasks.eccserver_refresh_state_task >
```

Fetch the state of the given ECC server.

This will contact the ECC server identified by the given primary key, fetch its state, and update the state in the database.

Parameters `eccserver_pk` (*int*) – The integer primary key of the `ECCServer` object in the database.

`attpcdaq.daq.tasks.eccserver_refresh_all_task`

`attpcdaq.daq.tasks.eccserver_refresh_all_task = <@task: attpcdaq.daq.tasks.eccserver_refresh_all_task of attpcdaq.daq.tasks>`
Fetch the state of all ECC servers.

This calls `eccserver_refresh_state_task()` for each ECC server in the database.

`attpcdaq.daq.tasks.eccserver_change_state_task`

`attpcdaq.daq.tasks.eccserver_change_state_task = <@task: attpcdaq.daq.tasks.eccserver_change_state_task of attpcdaq.daq.tasks>`
Change the state of an ECC server (make it perform a transition).

This will contact the ECC server identified by the given primary key and tell it to transition to the given target state. This is done by calling `change_state()` on the `ECCServer` object.

Parameters

- `eccserver_pk` (*int*) – The ECC server’s integer primary key.
- `target_state` (*int*) – The target state. Use one of the constants from the `ECCServer` class.

Checking remote status

<code>check_ecc_server_online_task</code>	Checks if the ECC server is online.
<code>check_ecc_server_online_all_task</code>	Check and update the state of all known ECC servers.
<code>check_data_router_status_task</code>	Checks whether the data router is online and if the staging directory is clean.
<code>check_data_router_status_all_task</code>	Check and update the state of all known data routers.

`attpcdaq.daq.tasks.check_ecc_server_online_task`

`attpcdaq.daq.tasks.check_ecc_server_online_task = <@task: attpcdaq.daq.tasks.check_ecc_server_online_task of attpcdaq.daq.tasks>`
Checks if the ECC server is online.

This is done by checking if the process is running via SSH. Specifically, the method `check_ecc_server_status()` of the `WorkerInterface` object is used.

Parameters `eccserver_pk` (*int*) – The primary key of the ECC server in the database.

`attpcdaq.daq.tasks.check_ecc_server_online_all_task`

`attpcdaq.daq.tasks.check_ecc_server_online_all_task = <@task: attpcdaq.daq.tasks.check_ecc_server_online_all_task of attpcdaq.daq.tasks>`
Check and update the state of all known ECC servers.

This calls `check_ecc_server_online_task()` for each ECC server.

attpcdag.daq.tasks.check_data_router_status_task

`attpcdag.daq.tasks.check_data_router_status_task = <@task: attpcdag.daq.tasks.check_data_router_status_task of attpcdag:0x7fe18425...`
Checks whether the data router is online and if the staging directory is clean.

This is done by checking if the process is running via SSH. Specifically, the method `check_data_router_status()` of the `WorkerInterface` object is used. Then, the staging directory is checked for GRAW files using `working_dir_is_clean()` from the same class.

Parameters `datarouter_pk (int)` – The primary key of the data router in the database.

attpcdag.daq.tasks.check_data_router_status_all_task

`attpcdag.daq.tasks.check_data_router_status_all_task = <@task: attpcdag.daq.tasks.check_data_router_status_all_task of attpcdag:0x7fe18425...`
Check and update the state of all known data routers.

This calls `check_data_router_status_task()` for each data router.

File organization

<code>organize_files_task</code>	Connects to the DAQ worker nodes to organize files at the end of a run.
<code>organize_files_all_task</code>	Organize files on all remote nodes.

attpcdag.daq.tasks.organize_files_task

`attpcdag.daq.tasks.organize_files_task = <@task: attpcdag.daq.tasks.organize_files_task of attpcdag:0x7fe18425...`
Connects to the DAQ worker nodes to organize files at the end of a run.

This is done via SSH using the method `organize_files()` of the `WorkerInterface` object.

Parameters

- `datarouter_pk (int)` – Integer primary key of the data source
- `experiment_pk (int)` – The primary key of the current experiment
- `run_pk (int)` – The primary key of the most recent run

attpcdag.daq.tasks.organize_files_all_task

`attpcdag.daq.tasks.organize_files_all_task = <@task: attpcdag.daq.tasks.organize_files_all_task of attpcdag:0x7fe18425...`
Organize files on all remote nodes.

This calls `organize_files_task()` for all data routers.

Parameters

- `experiment_pk (int)` – The primary key of the current experiment
- `run_pk (int)` – The primary key of the most recent run

Task scheduling

Some of the tasks above are best run automatically according to a schedule. Periodic tasks are supported by the Celery system, and are configured using the `CELERYBEAT_SCHEDULE` entry in the `attpcdaq.settings` module. This is a dictionary with the format shown in the example below.

```
CELERYBEAT_SCHEDULE = {
    'update-state-every-5-sec': {                                # A descriptive_
↪name for the task
        'task': 'attpcdaq.daq.tasks.eccserver_refresh_all_task', # The dotted name_
↪of the task, as a string
        'schedule': timedelta(seconds=5),                       # The interval_
↪between runs
    },
}
```


Symbols

`_get_soap_client()` (attpcdaq.daq.models.ECCServer method), 24
`_get_transition()` (attpcdaq.daq.models.ECCServer class method), 24

A

`as_xml()` (attpcdaq.daq.models.ConfigId method), 27

C

`calculate_overall_state()` (in module attpcdaq.daq.views.helpers), 38
`change_state()` (attpcdaq.daq.models.ECCServer method), 23
`check_data_router_status()` (attpcdaq.daq.workertasks.WorkerInterface method), 41
`check_data_router_status_all_task` (in module attpcdaq.daq.tasks), 44
`check_data_router_status_task` (in module attpcdaq.daq.tasks), 44
`check_ecc_server_online_all_task` (in module attpcdaq.daq.tasks), 43
`check_ecc_server_online_task` (in module attpcdaq.daq.tasks), 43
`check_ecc_server_status()` (attpcdaq.daq.workertasks.WorkerInterface method), 41
`choose_config()` (in module attpcdaq.daq.views.pages), 32
`ConfigId` (class in attpcdaq.daq.models), 26

D

`DataRouter` (class in attpcdaq.daq.models), 25
`DataSource` (class in attpcdaq.daq.models), 25
`duration` (attpcdaq.daq.models.RunMetadata attribute), 30
`duration_string` (attpcdaq.daq.models.RunMetadata attribute), 31

E

`easy_setup()` (in module attpcdaq.daq.views.pages), 34
`EasySetupPage` (class in attpcdaq.daq.views.pages), 33
`ECCServer` (class in attpcdaq.daq.models), 22
`eccserver_change_state_task` (in module attpcdaq.daq.tasks), 43
`eccserver_refresh_all_task` (in module attpcdaq.daq.tasks), 43
`eccserver_refresh_state_task` (in module attpcdaq.daq.tasks), 42
`Experiment` (class in attpcdaq.daq.models), 28
`experiment_settings()` (in module attpcdaq.daq.views.pages), 33

F

`find_data_router()` (attpcdaq.daq.workertasks.WorkerInterface method), 40
`from_xml()` (attpcdaq.daq.models.ConfigId class method), 27

G

`get_data_link_xml()` (attpcdaq.daq.models.DataSource method), 26
`get_data_link_xml_from_clients()` (attpcdaq.daq.models.ECCServer method), 23
`get_data_router_statuses()` (in module attpcdaq.daq.views.helpers), 39
`get_ecc_server_statuses()` (in module attpcdaq.daq.views.helpers), 38
`get_graw_list()` (attpcdaq.daq.workertasks.WorkerInterface method), 41
`get_status()` (in module attpcdaq.daq.views.helpers), 39

I

`is_running` (attpcdaq.daq.models.Experiment attribute), 29

L

`latest_run` (attpcdaq.daq.models.Experiment attribute), 29

M

Measurement (class in `attpcdaq.daq.models`), [31](#)

N

`next_run_number` (`attpcdaq.daq.models.Experiment` attribute), [29](#)

O

Observable (class in `attpcdaq.daq.models`), [31](#)

`organize_files()` (`attpcdaq.daq.workertasks.WorkerInterface` method), [41](#)

`organize_files_all_task` (in module `attpcdaq.daq.tasks`), [44](#)

`organize_files_task` (in module `attpcdaq.daq.tasks`), [44](#)

R

`refresh_configs()` (`attpcdaq.daq.models.ECCServer` method), [24](#)

`refresh_state()` (`attpcdaq.daq.models.ECCServer` method), [24](#)

`refresh_state_all()` (in module `attpcdaq.daq.views.api`), [36](#)

RunMetadata (class in `attpcdaq.daq.models`), [30](#)

S

`save()` (`attpcdaq.daq.models.Experiment` method), [30](#)

`set_observable_ordering()` (in module `attpcdaq.daq.views.api`), [37](#)

`show_log_page()` (in module `attpcdaq.daq.views.pages`), [33](#)

`source_change_state()` (in module `attpcdaq.daq.views.api`), [35](#)

`source_change_state_all()` (in module `attpcdaq.daq.views.api`), [35](#)

`start_run()` (`attpcdaq.daq.models.Experiment` method), [29](#)

`status()` (in module `attpcdaq.daq.views.pages`), [32](#)

`stop_run()` (`attpcdaq.daq.models.Experiment` method), [30](#)

T

`tail_file()` (`attpcdaq.daq.workertasks.WorkerInterface` method), [42](#)

W

WorkerInterface (class in `attpcdaq.daq.workertasks`), [40](#)

`working_dir_is_clean()` (`attpcdaq.daq.workertasks.WorkerInterface` method), [41](#)