Automatic Run Processing (ARP)

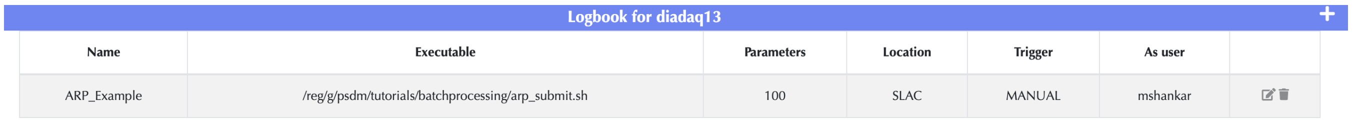
The Automatic Run Processor (or ARP, for short, because I want that to catch on) is a web service that allows for automatic workflows and for the easier submission of batch jobs via a web interface. A script that submits the batch job (to allow for more customization in this command) is all that is needed for this system to work.

1) Webpage

To use this system, choose an experiment from https://pswww.slac.stanford.edu/lgbk/lgbk/experiments**.**

1.1) Workflow Definitions

Under the **Workflow** dropdown, select **Definitions**. The **Workflow Definitions** tab is the location where scripts are defined; scripts are registered with a unique name. Any number of scripts can be registered. To register a new script, please click on the + button on the top right hand corner.



Logbook for diadaq13

Name	Executable	Parameters	Location	Trigger	As user	
ARP_Example	/reg/g/psdm/tutorials/batchprocessing/arp_submit.sh	100	SLAC	MANUAL	mshankar	

1.1.1) Name

A unique name given to a registered script; the same script can be registered under different names with different parameters.

1.1.2) Executable

The absolute path to the batch script. This script must contain the batch job submission command (**sbatch** for SLURM). It gives the user the ability to customize the the batch submission. Overall, it can act as a wrapper for the code that will do the analysis on the data along with submitting the job.

1.1.3) Parameters

The parameters that will be passed to the executable as command line arguments. These parameters can be used as parameters to the **sbatch** command to specify the queue, number of cores etc. Or; they can be used to customize the script execution. In addition, details of the batch job are made available as as environment variables.

- EXPERIMENT - The name of the experiment; in the example shown above, **diadaq13**.
- RUN_NUM - The run number for the job.
- ARP_UPDATE_COUNTERS - This is a URL that can be used to update the progress of the job. These updates are also automatically reflected in the UI. In previous releases, this was called JID_UPDATE_COUNTERS.
- ARP_JOB_ID - The id for this job execution; this is an internal identified used in API calls to the other data management systems for example, to update counters or in AirFlow integrations.
- ARP_ROOT_JOB_ID - If using Airflow or other workflow engines; this is the identifier of the initial job in the DAG.
- ARP_LOCATION - The data management location that this job is running at; for example, S3DF or NERSC.
- ARP_SLURM_ACCOUNT - The SLURM account to be used in sbatch calls (if applicable).

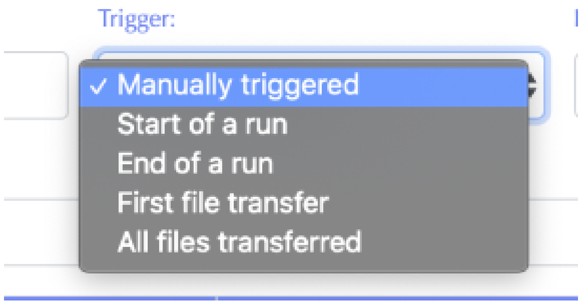
1.1.4) Location

This defines where the analysis is done. While many experiments prefer to use the SLAC psana cluster (SLAC) or the SRCF (SRCF_FFB) to perform their analysis, others prefer to use HPC facilities like NERSC to perform their analysis.

1.1.4) Trigger

This defines the event that in the data management system that kicks off the job submission.

- Manually triggered - The user will manually trigger the job in the **Workflow Control** tab.
- Start of a run - When the DAQ starts a new run.
- End of a run - When the DAQ closes a run.
- First file transfer - When the data movers indicate that the first registered file for a run has been transferred and is available at the job location.
- All files transferred - When the data movers indicate that all registered files for a run have been transferred and are available at the job location.



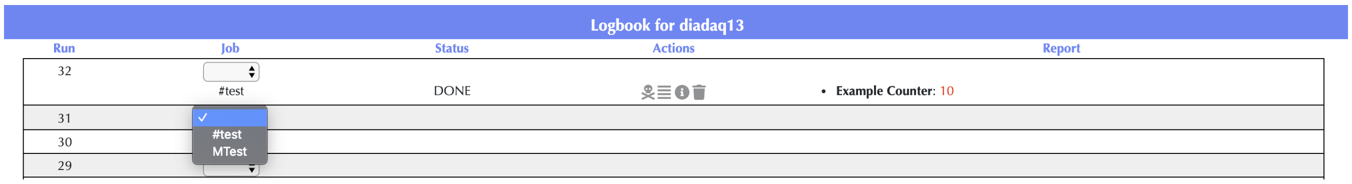
1.1.5) As user

If the job is automatically triggered, it will be executed as this user. If the job is manually triggered; it will be executed as the user triggering the job manually. This is set when creating the job definition and cannot be changed.

1.1.6) Edit/delete job
Use the edit/trash icons to edit a job definition or to delete a job definition

1.2) Workflow Control

Under the **Workflow** dropdown, select **Control** to create and check the status of your analysis jobs. The **Control** tab is where job definitions defined in the **Definitions** tab may be applied to experiment runs. An entry is automatically created for jobs that are triggered automatically. To manually trigger a job, in the drop-down menu of the **Job** column, select the job. A job can be triggered any number of times; each execution has a separate row in the UI.



Logbook for diadaq13

Run	Job	Status	Actions	Report
32	#test	DONE		• **Example Counter**: 10
31				
30				
29				

1.2.1) Status

These are the different statuses that a job can have -

- START - Pending submission to the HPC workload management infrastructure.
- SUBMITTED - The job has been submitted to the HPC workload management infrastructure. A job may stay in the SUBMITTED for some time depending on how busy the queues are.
- RUNNING - The job is currently running. One can get job details and the log file. The job can also potentially be killed.
- EXITED - The job has finished unsuccessfully. The log files may have some additional information.
- DONE - The job has finished successfully. The job details and log files may be available; most HPC workload management systems delete this information after some time.

1.2.2) Actions

There are four different actions which can be applied to a script. They do the following if pressed:

- Attempt to kill the job. A green success statement will appear near the top-right of the page if the job is killed successfully and a red failure statement will appear if the job is not killed successfully.

- Returns the log file for the job. If there is no log file or if no log file could be found, it will return blank.

- Returns details for the current job by invoking the the appropriate job details command in the HPC workload management infrastructure.

🗑 - Delete the job execution from the run. **Note:** this does not kill the job, it only removes it from the webpage.

1.2.3) Report

This is a customizable column which can be used by the script executable to report progress. The script executable reports progress by posting JSON to a URL that is available as the environment variable **JID_UPDATE_COUNTERS**.

For example, to update the status of the job using bash, one can use

```
curl -s -XPOST ${JID_UPDATE_COUNTERS} -H "Content-Type: application/json" -d '[ {"key": "<b>LoopCount</b>",
"value": "'"${i}"'" } ]'
```

In Python, one can use

```
import os
import requests
requests.post(os.environ["JID_UPDATE_COUNTERS"], json=[ {"key": "<b>LoopCount</b>", "value": "75" } ])
```

2) Examples.

2.1) arp_submit.sh

The executable script used in the workflow definition should be used primarily to set up the environment etc and submit the analysis script to the HPC workload management infrastructure. For example, a simple executable script that uses SLURM's sbatch to submit the analysis script is available here - */reg/g/psdm/tutorials/batchprocessing/arp_submit.sh*

```
#!/bin/bash

source /reg/g/psdm/etc/psconda.sh
ABS_PATH=/reg/g/psdm/tutorials/batchprocessing
sbatch --nodes=2 --partition=psanaq --time=5 --output="arp_example_${RUN_NUM}_%j.log"  $ABS_PATH/arp_actual.py
"$@"
```

This script will submit */reg/g/psdm/tutorials/batchprocessing/arp_actual.py*. */reg/g/psdm/tutorials/batchprocessing/arp_actual.py* will be passed the parameters as command line arguments and will inherit the EXPERIMENT, RUN_NUM and JID_UPDATE_COUNTERS environment variables.
Log files:
If the --output parameter is not specified to sbatch, then SLURM will store the log output in */reg/d/psdm/dia/diadaq13/scratch/<slurm_job_id>.out*
In the example above, the log output will be sent to to the default working folder for the job; which is the scratch folder but the file name will be generated using the run number and the job id. For example, the log file for run 25 job id 409327. will be send to */reg/d/psdm/dia/diadaq13/scratch/arp_example_25_409327.log*
To avoid cluttering the scratch folder, one can use an absolute path in the --output command to specify an alternate location for the job log files. See the "filename pattern" in the sbatch man page for more details.

2.2) arp_actual.py

This Python script is the code that will do analysis and whatever is necessary on the run data. Since this is just an example, the Python script, **arp_actual.py**, doesn't get that involved. It is shown below.

```
#!/usr/bin/env python
import os
import sys
import requests
import time
import datetime
import logging

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

logger.debug("In the arp_actual script - current time is %s", datetime.datetime.now().strftime("%c"))

for k, v in sorted(os.environ.items()):
    logger.debug("%s=%s", k, v)

## Fetch the URL to post progress updates
update_url = os.environ.get('JID_UPDATE_COUNTERS')
logger.debug("The URL to post updates is %s", update_url)

# These are the parameters that are passed in
logger.debug("The parameters passed into the script are %s", " ".join(sys.argv))

loop_count = 20
try:
    loop_count = int(sys.argv[1])
except:
    pass

## Run a loop, sleep a second, then POST
for i in range(loop_count):
    time.sleep(1)
    logger.debug("Posting for step %s", i)
    requests.post(update_url, json=[{"key": "<strong>Counter</strong>", "value" : "<span style='color: red'>{0}<
/span>".format(i+1)}, {"key": "<strong>Current time</strong>", "value": "<span style='color: blue'>{0}</span>".
format(datetime.datetime.now().strftime("%c"))}])


logger.debug("Done with job execution")
```

2.3) Log File Output

The **logger.debug** statements are sent to the job's log file. Note, one can form **sbatch** commands where the log output is not sent to a logfile and is instead sent as an email. Part of an example log file output is shown below.

```
DEBUG:__main__:In the arp_actual script - current time is Thu Apr 16 11:12:40 2020
...
DEBUG:__main__:EXPERIMENT=diadaq13
...
DEBUG:__main__:JID_UPDATE_COUNTERS=https://pswww.slac.stanford.edu/ws/jid_slac/jid/ws/replace_counters
/5e98a01143a11e512cb7c8ca
...
DEBUG:__main__:RUN_NUM=26
...
DEBUG:__main__:The parameters passed into the script are /reg/g/psdm/tutorials/batchprocessing/arp_actual.py 100
DEBUG:__main__:Posting for step 0
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): pswww.slac.stanford.edu:443
DEBUG:urllib3.connectionpool:https://pswww.slac.stanford.edu:443 "POST /ws/jid_slac/jid/ws/replace_counters
/5e98a01143a11e512cb7c8ca HTTP/1.1" 200 195
DEBUG:__main__:Posting for step 1
...
DEBUG:__main__:Done with job execution
...
```

3.0 Frequently Asked Questions (FAQ)

Is it possible to submit more than one job per run?

- Yes, each run can accept multiple hashtags.

Can a submitted job submit other subjobs?

- Yes, in a standard LSF/SLURM fashion, BUT the ARP will not know about the subjobs. Only jobs submitted through the ARP webpage are known to the ARP.

When using the 'kill' option, how does ARP know which jobs to kill?

- The ARP keeps track of the hashtags for each run and the associated LSF/SLURM jobid. That information allows the ARP to kill jobs.

As far as I understand there is a json entry for each line which stores info, can one access this json entry somehow?

- The JSON values are displayed in the ARP webpage automatically. To access them programmatically, use the kerberos endpoint. See API access to the LCLS2 eLog for more details.