**Logging in cronjobs**

All cronjobs should have a minimum amount of custom logging, in addition to the automated logging we have in place.

This custom logging should use a new Laravel log channel setup according to the documentation here: [link to laravel docs on channel setup]

The reason for this is to help clearly separate each cronjob's log output from other sources of log output when viewing combined logs, and in order to make it easier for us to have each job written to a different stack of output drivers.

For now, each cronjob should log to the "stack" output driver.

Note that this article currently focuses mainly on PHP projects, but please follow the same formats in Deno and Python backend code as well to the extent this is possible. All of our code needs to be debuggable.

**Key points**

1. Remember, the point of this logging guidance is to create *extremely* verbose log output. The reason for this is that it is impossible to debug a problem without enough information, and often the only way to gather than information from the past is to always log it.

2. We will not send all of this log output to Datadog, but we may collect it using operating system-level functionality in a central location for a relatively brief period of time (up to a few weeks at most likely).

3. This is guidance only, but the formats used here should be followed as closely as possible — and *new formats should be discussed and added whenever needed*!

4. In PHP projects: This log output should *always* use the Log facade and *always* use the debug() log level. Other types of log output may use other levels, but we should *always* use the Laravel Log facade in PHP projects.

**Places to add log output**

Logging should be added to the following locations, which together form the bulk of the code path for most cronjobs.

Namespaces:

- App\Console\Commands
- 
- App\Http\Controllers
- 
- App\Jobs
- App\Services
- App\Repositories

- Log points:

  - At the **start** of every method implemented on a class within the above namespaces:
    `$callId = uniqid(getmypid(), true);`
    `Log::debug(__FILE__."#".__LINE__."::".__FUNCTION__." BEGIN [$callId] (".json_encode([param1 => $param1, param2 => $param2, param3 => $param3, ...etc...]).")");`
    This creates a very readable and easy to use line of log output. For example, a line of log output from this call might look like:

`[2022-12-12 15:58:49] local.DEBUG: OptimizeUserImage.php#84::handle BEGIN [aaaaaa1111111111] ({"uniqJobId": "39jd3.d39d", "influencerProductId": 100, ...})`

- Using this line of output, it is trivial to find the location where this log line was generated, and to see what parameters are passed in to the method.
- The __FILE__ constant is used instead of the __CLASS__ constant, because the location where a class is implemented is not always readily apparent — and not all code in Laravel projects is implemented in classes (for instance, global helper functions are simple functions).
- For this reason we also do not use the __METHOD__ constant (which would also include the self::CLASS_NAME value).
- In cases where the class is of primary is important, of course, feel free to adjust this format or include it before or after the other parts of the output.
- Keeping the internal format consistent may be helpful in increasing readability however, so keep this in mind, while including all the information you would need to debug your code in the absense of any other data.

- At the **end** of every method implemented on a class within the above locations:
  `Log::debug(__FILE__."#".__LINE__."::".__FUNCTION__." END [$callId] = {".json_encode($result)."}");`
  `return $result`
  - Example output might be:
    `[2022-12-12 15:58:50] local.DEBUG: OptimizeUserImage.php#120::handle END [aaaaaa1111111111] = {true}`
  - Obviously in the case of a void function, no $result will exist to be logged, so you can leave out these parts.
  - The benefit of the $callId variable is now clear — as log output is always intermingled with other log calls, we need a unique ID in order to associate the return line with the starting line. The function of the BEGIN and END tokens helps us know which line is which.
  - We wrap the return result in curly brackets in order to ensure that we can differentiate between an absent value or other formatting error. An object returned here would result in double curly brackets.
  - Do **not** log very large objects, instead log only key parts of each object in order to identify what object it was. Also log any important changes made to the object within the function/method itself, see next section.
- At key points within each method/function, log the important changes made by the code and actions with side effects taken (saving, deleting, etc.)
  - For instance, use this format to log changes made to an object:
    `Log::debug(__FILE__."#".__LINE__."::".__FUNCTION__." [$callId]: old($fieldName)={$oldValue}, new($fieldName)={$newValue}");`
  - This might result in:
    `[2022-12-12 15:58:50] local.DEBUG: OptimizeUserImage.php#120::handle [aaaaaa1111111111]: old_first_name={Some value}, new_first_name={New value}"`
    - Again, callId is important to include so that we can trace which call with which parameters (and later, which result) is associated with this change.

**Example code:**

[add example code]