

# Batch Processing and the Drupal Queue System

## TABLE OF CONTENTS

- [Batch Processing Using the Batch API](#)
  - [Overview](#)
  - [Using the Batch API with a form](#)
  - [Using the Batch API from a controller](#)
  - [Using the Batch API with hook\\_update](#)
  - [Important rules about functions when using Batch API](#)
  - [Looking at the source](#)
    - [Passing parameters to the functions in a batch operation](#)
- [Queue System](#)
- [Resources](#)

views 398

## Batch Processing Using the Batch API

### Overview

The Batch API provides very useful functionality that lets you do work by breaking it into pieces to avoid PHP timeouts, etc. Usually, you'll do this by creating a group of node id's (using `array_chunk()`) to be processed and use the batch API to process those arrays (chunks) of ids. You can also provide code that will let it figure out how much work to do and stop itself.

In addition, you create a function to handle things once all the chunks are complete. You can also give the Batch API a bunch of work to do and have it figure out for itself when it is finished.

The Batch API uses the Drupal Queue system, allowing it to pick up where it left off in case of problems.

You can use the Batch API in controllers, forms, hook updates, and Drush commands. The implementation of each one is slightly different, as you can see in the examples.

Most often you start a batch from a form where you fill in some options and click a button. You can also run a batch on a controller where the batch operation starts when you point the browser at a URL. Batch operations in Drush commands are started when you type the command in the terminal.

In the code: For a drush command, you use `drush_backend_batch_process()` to kick off the batch (from `DirtSalesforceController.php`):

```
if ($browser === FALSE) {  
  drush_backend_batch_process();  
}
```

Here is an explanation from the source of what it does:

- \* Process a **Drupal** batch by spawning multiple **Drush** processes.
- \*
- \* This function will include the correct batch engine for the current major version of **Drupal**, and will make use of the `drush_backend_invoke` system to spawn multiple worker threads to handle the processing of the current batch, while keeping track of available memory.
- \*
- \* The batch system will process as many batch sets as possible until the entire batch has been completed or 60% of the available memory has been used.
- \*
- \* This function is a drop in replacement for the existing `batch_process()` function of Drupal.

Although it will be covered in more detail in the future, it is useful to know that there is now a batch builder: The [BatchBuilder class](#) for batch API. It provides a more streamlined object-oriented approach to creating batches.

## Using the Batch API with a form

This example replaces a multivalue field with some new values processing 10 nodes at a time. The decision to process 10 at a time is arbitrary, but be aware that the more nodes you process at a time, the higher the possibility that the process will time out causing the batch will fail.

The form example is in the accompanying source and is accessed at <https://d9book.ddev.site/batch-examples/batchform>

The source file is at `web/modules/custom/batch_examples/src/Form/BatchForm.php` and is presented in pieces below:

Here is a simple form with a button used to kick off the batch operation.

```
/**
 * {@inheritdoc}
 */

public function buildForm(array $form, FormStateInterface $form_state): array {
    $form['message'] = [
        '#markup' => $this->t('Click the button below to kick off the batch!'),
    ];
    $form['actions']['#type'] = 'actions';
    $form['actions']['submit'] = [
        '#type' => 'submit',
        '#value' => $this->t('Run Batch'),
    ];

    return $form;
}
```

The `submitForm()` method calls `updateEventPresenters()`.

```
/**
 * {@inheritdoc}
 */

public function submitForm(array &$form, FormStateInterface $form_state): void {
    $this->updateEventPresenters();
    $this->messenger()->addStatus($this->t('The message has been sent.'));
    $form_state->setRedirect('<front>');
}
```

The route is:

```

batch_examples.batch:
  path: '/batch-examples/batchform'
  defaults:
    _title: 'Batch Form'
    _form: 'Drupal\batch_examples\Form\BatchForm'
  requirements:
    _permission: 'access content'

```

Here is the `updateEventPresenters()` method. Notice the `$operations` array, which contains the function to call to do the work of each batch as well as the list of nids to process.

```

function updateEventPresenters(): void {
    $query = \Drupal::entityQuery('node')
        ->condition('status', 1)
        ->condition('type', 'event')
        ->sort('title', 'ASC')
        ->accessCheck(TRUE);
    $nids = $query->execute();

    // Create batches.
    $chunk_size = 10;
    $chunks = array_chunk($nids, $chunk_size);
    $num_chunks = count($chunks);

    // Submit batches.
    $operations = [];
    for ($batch_id = 0; $batch_id < $num_chunks; $batch_id++) {
        $operations[] = [
            '\Drupal\batch_examples\Form\BatchForm::exampleProcessBatch',
            [
                $batch_id+1,
                $chunks[$batch_id]],
        ];
    }
    $batch = [
        'title' => $this->t('Updating Presenters'),
        'init_message' => $this->t('Starting to process events.'),
        'progress_message' => $this->t('Completed @current out of @total batches.'),
        'finished' => '\Drupal\batch_examples\Form\BatchForm::batchFinished',
        'error_message' => $this->t('Event processing has encountered an error.'),
        'operations' => $operations,
    ];
    batch_set($batch);
}

```

Here is the method that actually does the work. Most of the code is for information reporting. The actual work is in the `foreach $nids as $nid` loop:

```

public static function exampleProcessBatch(int $batch_id, array $nids, array &$amp;context): void {
    if (!isset($context['sandbox']['progress'])) {
        $context['sandbox']['progress'] = 0;
        $context['sandbox']['current_node'] = 0;
        $context['sandbox']['max'] = 0;
    }
    if (!isset($context['results']['updated'])) {
        $context['results']['updated'] = 0;
        $context['results']['skipped'] = 0;
        $context['results']['failed'] = 0;
        $context['results']['progress'] = 0;
    }

    // Keep track of progress.
    $context['results']['progress'] += count($nids);
    $context['results']['process'] = 'Import request files';
    // Message above progress bar.
    $context['message'] = t('Processing batch #@batch_id batch size @batch_size for total @count items.', [
        '@batch_id' => number_format($batch_id),
        '@batch_size' => number_format(count($nids)),
        '@count' => number_format($context['sandbox']['max']),
    ]);

    foreach ($nids as $nid) {
        $filename = '';
        /** @var \Drupal\node\NodeInterface $event_node */
        $event_node = Node::load($nid);
        if ($event_node) {
            $array = ['Mary Smith', 'Fred Blue', 'Elizabeth Queen'];
            shuffle($array);
            $event_node->field_presenter = $array;
            $event_node->save();
        }
    }
}

```

The Form API will take care of getting the batches executed. If you aren't using a form, you use `batch_process()` like the line shown below. For this method, you specify any valid alias and the system will redirect to that alias after the batch completes.

```

// node/1 should be a valid node.
return batch_process('node/1');

```

#### NOTE

Also you can set up a `$batch` array with a title and a progress message with some variables that will get displayed.

You specify a `finished` index, which identifies a function to call after the batch is finished processing, as in the example below.

```

'finished' => '\Drupal\batch_examples\Form\BatchForm::batchFinished',

```

Here is the `batchFinished()` method, which displays and logs the results.

```

use Symfony\Component\HttpFoundation\RedirectResponse;

/**
 * Handle batch completion.
 *
 * @param bool $success
 *   TRUE if all batch API tasks were completed successfully.
 * @param array $results
 *   An array of processed node IDs.
 * @param array $operations
 *   A list of the operations that had not been completed.
 * @param string $elapsed
 *   Batch.inc kindly provides the elapsed processing time in seconds.
 */
public static function batchFinished(bool $success, array $results, array $operations, string $elapsed): RedirectResponse {
    $messenger = \Drupal::messenger();
    if ($success) {
        $messenger->addMessage(t('@process processed @count nodes, skipped @skipped, updated @updated, failed @failed in @elapsed.', [
            '@process' => $results['process'],
            '@count' => $results['progress'],
            '@skipped' => $results['skipped'],
            '@updated' => $results['updated'],
            '@failed' => $results['failed'],
            '@elapsed' => $elapsed,
        ]));
        \Drupal::logger('d9book')->info(
            '@process processed @count nodes, skipped @skipped, updated @updated, failed @failed in @elapsed.', [
                '@process' => $results['process'],
                '@count' => $results['progress'],
                '@skipped' => $results['skipped'],
                '@updated' => $results['updated'],
                '@failed' => $results['failed'],
                '@elapsed' => $elapsed,
            ]);
    }
    else {
        // An error occurred.
        // $operations contains the operations that remained unprocessed.
        $error_operation = reset($operations);
        $message = t('An error occurred while processing %error_operation with arguments: @arguments', [
            '%error_operation' => $error_operation[0],
            '@arguments' => print_r($error_operation[1], TRUE),
        ]);
        $messenger->addError($message);
    }
    // Optionally redirect back to the form.
    return new RedirectResponse('/batch-examples/batchform');
}

```

## Using the Batch API from a controller

The Batch API is often used in connection with forms. If you're using a page callback, you will need to setup all the items, submit them to the batch API, and then call `batch_process()` with a url as the argument.

```
return batch_process('node/1');
```

After the batch is complete, Drupal will redirect you to that url. E.g. `/node/1`

In this example of a processing function, you can see error handling, logging, and tracking while retrieving files from a remote source. This is fairly common when moving data between systems. The rest of the code is almost identical to the previous example.

```
public static function fileImportProcessBatch(int $batch_id, array $nids, array &$context): void {
    if (!isset($context['sandbox']['progress'])) {
        $context['sandbox']['progress'] = 0;
        $context['sandbox']['current_node'] = 0;
        $context['sandbox']['max'] = 0;
    }
    if (!isset($context['results']['updated'])) {
        $context['results']['updated'] = 0;
        $context['results']['skipped'] = 0;
        $context['results']['failed'] = 0;
        $context['results']['progress'] = 0;
    }
    // Total records to process for all batches.
    if (empty($context['sandbox']['max'])) {
        $query = \Drupal::entityQuery('node')
            ->condition('status', 1)
            ->condition('type', 'opinion_request');
        $total_nids = $query->execute();
        $context['sandbox']['max'] = count($total_nids);
    }
}
```

*// Keep track of progress.*

```
$context['results']['progress'] += count($nids);
```

```
$context['results']['process'] = 'Import request files';
```

*// Message above progress bar.*

```
$context['message'] = t('Processing batch #@batch_id batch size @batch_size for total @count items.', [
    '@batch_id' => number_format($batch_id),
    '@batch_size' => number_format(count($nids)),
    '@count' => number_format($context['sandbox']['max']),
]);
```

```
$fileRepository = \Drupal::service('file.repository');
```

```
foreach ($nids as $nid) {
```

```
    $filename = '';
```

```
    $request_node = Node::load($nid);
```

```
    if ($request_node) {
```

```
        $file_id = $request_node->get('field_request_file')->target_id;
```

```
        if (!empty($file_id)) {
```

*// confirm that the file exists.*

```
        $file = File::load($file_id);
```

```
        if ($file) {
```

```
            $uri = $file->getFileUri();
```

```
            if (file_exists($uri)) {
```

```
                $context['results']['skipped']++;
```

```
                continue;
```

```
            }
```

```
        }
```

```
    }
```

*//Skip retrieving file if there is no request date.*

```
$request_date = $request_node->get('field_request_date')->value;
```

```
if (empty($request_date)) {
```

```
    $context['results']['skipped']++;
```

```
    continue;
```

```

}

$source_url = $request_node->get('field_pdf_file_with_path')->value;
if ($source_url == "missing") {
    $context['results']['skipped']++;
    continue;
}

if (!empty($source_url)) {
    $filename = basename($source_url);
    if (empty($filename)) {
        \Drupal::logger('oag_opinions')
            ->error('file_import - Error retrieving file - invalid filename in' . $source_url);
        $context['results']['skipped']++;
        continue;
    }

    $file_contents = @file_get_contents($source_url);
    if ($file_contents === FALSE) {
        \Drupal::logger('oag_opinions')
            ->error('file_import - Error retrieving file ' . $source_url);
        \Drupal::messenger()->addError(t('Error retrieving file %filename.', ['%filename' => $source_url]), FALSE);
        $context['results']['failed']++;
        continue;
    }

    $destination = "public://" . $filename;
    $file = null;
    try {
        $file = $fileRepository->writeData($file_contents, $destination, FileSystemInterface::EXISTS_REPLACE);
    }
    catch (FileNotFoundException $e) {
        \Drupal::logger('oag_opinions')->error('file_import - Error saving file ' . $destination);
        \Drupal::messenger()->addError(t('Error saving file %filename', ['%filename' => $destination]), FALSE);
        $context['results']['failed']++;
        continue;
    }

    if (!$file) {
        $context['results']['failed']++;
        continue;
    }

    $fid = $file->id();
    $request_node->set('field_request_file', $fid);
    // $request_node->field_request_file->target_id = $fid;

    $request_node->save();
    $context['results']['updated']++;
}
}
}
}

```

Here is the code that creates the batches and submits them.

```

public function summaryImport() {
    $this->summaryCreateBatches();
    return batch_process('/admin/content');
}

```

## Using the Batch API with hook\_update

If you want to update the default value of a field for all nodes using the Batch API and hook\_update\_N checkout the following links:

- [Using the Batch API and hook\\_update\\_N in Drupal 8](#)

- [Drupal API | batch\\_example\\_update\\_8001 | batch\\_example.install](#)

## Important rules about functions when using Batch API

All batch functions must be public static functions and all functions calling those must be explicitly namespaced like:

```
$nid = \Drupal\dirty_salesforce\Controller\DirtySalesforceController::lookupCommodityItem($commodity_item_id);
```

You can't use `$this->my_function` even if they are in the same class. Grab the namespace from the top of the PHP file you are using. In this case:

```
namespace Drupal\dirty_salesforce\Controller;
```

You can however refer to the functions with `self::` e.g.

```
$node_to_update_dirty_contact_nid = self::getFirstRef($node_to_update, 'field_sf_dirty_contact_ref');
```

## Looking at the source

The [source code for the Batch API](#) is really well commented and worth reading.

### PASSING PARAMETERS TO THE FUNCTIONS IN A BATCH OPERATION

In this file <https://git.drupalcode.org/project/drupal/-/blob/10.1.x/core/includes/form.inc#L562-678>, there is an example batch that defines two operations that call `my_function_1` and `my_function_2`. Notice how parameters can be passed to `my_function_1` separated by commas. From <https://git.drupalcode.org/project/drupal/blob/8.7.8/core/includes/form.inc#L570>:

```
* Example:
* @code
* $batch = array(
*   'title' => t('Exporting'),
*   'operations' => array(
*     array('my_function_1', array($account->id(), 'story')),
*     array('my_function_2', array()),
*   ),
*   'finished' => 'my_finished_callback',
*   'file' => 'path_to_file_containing_my_functions',
* );
* batch_set($batch);
* // Only needed if not inside a form __submit handler.
* // Setting redirect in batch_process.
* batch_process('node/1');
```

### NOTE

To execute the batch, the example shows a call to `batch_process('node/1')`. This could be any valid url alias e.g., `/admin/content`.

So here are the arguments for `my_function_1`:

```
* function my_function_1($uid, $type, &$amp;context) {
```

You call the batch finished function with the following arguments:



```

/**
 * Handle batch completion.
 *
 * @param bool $success
 *   TRUE if all batch API tasks were completed successfully.
 * @param array $results
 *   An array of processed node IDs. - or whatever you put in $context['results'][]
 * @param array $operations
 *   A list of the operations that had NOT been completed.
 * @param $elapsed
 *   Elapsed time for processing in seconds.
 */
public static function batchFinished($success, array $results, array $operations, $elapsed) {

```

The results are displayed:

```

$messenger = \Drupal::messenger();
if ($success) {
  $messenger->addMessage(t('Processed @count nodes in @elapsed.', [
    '@count' => count($results),
    '@elapsed' => $elapsed,
  ]));
}

```

You can load the \$results array with all sorts of interesting data, such as:

```

$context['results']['skipped'] = $skipped;
$context['results']['updated'] = $updated;

```

Batch API provides a nice way to display detailed results using code like:

```

$messenger->addMessage(t('Processed @count nodes, skipped @skipped, updated @updated in @elapsed.', [
  '@count' => $results['nodes'],
  '@skipped' => $results['skipped'],
  '@updated' => $results['updated'],
  '@elapsed' => $elapsed,
]));

```

Which produce the following output:

```

Processed 50 nodes, skipped 45, updated 5 in 3 sec.

```

You can display an informative message above the progress bar this way.

I filled in the \$context['sandbox']['max'] with a value, but I could have used \$context['sandbox']['whole-bunch'] or any variable here.

```

$context['sandbox']['max'] = count($max_nids);

```

An informative message above the progress bar using number\_format puts commas in the number if it is over 1,000.

```

$context['message'] = t('Processing total @count nodes',
  ['@count' => number_format($context['sandbox']['max'])]
);

```

Also you could show something about which batch number is running.

```

$operation_details = 'Yoyoma';

$id = 9;

$context['message'] = t('Running Batch "@id" @details',
  ['@id' => $id, '@details' => $operation_details]
);

```

You do have to provide your own info for the variables.

You can also stop the batch engine yourself with something like this. If you don't know beforehand how many records you need to process, you could use code like this.

```

// Inform the batch engine that we are not finished,
// and provide an estimation of the completion level we reached.
if ($context['sandbox']['progress'] != $context['sandbox']['max']) {
  $context['finished'] = ($context['sandbox']['progress'] >= $context['sandbox']['max']);
}

```

## Queue System

From [Alan Saunders article](#) on December 2021:

A queue is simply a list of stuff that gets worked through one by one, one analogy could be a conveyor belt on a till in a supermarket, the cashier works through each item on the belt to scan them.

Queues are handy in Drupal for chunking up large operations, like sending emails to many people. By using a queue, you are trying to avoid overloading the servers resources which could cause the site to go offline until the resources on the server are free'd up.

From [Sarthak TTN](#) on Feb 2017:

This is the `submitForm()` which creates an item and puts it in the queue.

```

/**
 * {@inheritdoc}
 */

public function submitForm(array &$form, FormStateInterface $form_state): void {
  /** @var QueueFactory $queue_factory */
  $queue_factory = \Drupal::service('queue');

  /** @var QueueInterface $queue */
  $queue = $queue_factory->get('email_processor');

  $item = new \stdClass();
  $item->username = $form_state->getValue('name');
  $item->email = $form_state->getValue('email');
  $item->query = $form_state->getValue('query');

  $queue->createItem($item);
}

```

Then you create a Queue Worker that implements `ContainerFactoryPluginInterface` and in the `processItem()` it processes a single item from the queue.

```

namespace Drupal\my_module\Plugin\QueueWorker;

use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
use Drupal\Core\Queue\QueueWorkerBase;
use Drupal\Core\Mail\MailManager;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * {@inheritdoc}
 */

class EmailEventBase extends QueueWorkerBase implements ContainerFactoryPluginInterface {

    /**
     * @param Drupal\Core\Mail\MailManager $mail
     *   The mail manager.
     */

    public function __construct(protected MailManager $mail) {}

    /**
     * {@inheritdoc}
     */

    public static function create(ContainerInterface $container, array $configuration, $plugin_id, $plugin_definition) {
        return new static($container->get('plugin.manager.mail'));
    }

    /**
     * Processes a single item of Queue.
     */

    public function processItem($data) {
        $params['subject'] = t('query');
        $params['message'] = $data->query;
        $params['from'] = $data->email;
        $params['username'] = $data->username;
        $to = \Drupal::config('system.site')->get('mail');
        $this->mail->mail('my_module', 'query_mail', $to, 'en', $params, NULL, true);
    }
}

```

Then you'll need a cronEventProcessor which in annotation tells cron how often to run the job:

```

namespace Drupal\my_module\Plugin\QueueWorker;

/**
 *
 * @QueueWorker(
 *   id = "email_processor",
 *   title = "My custom Queue Worker",
 *   cron = {"time" = 10}
 * )
 */

class CronEventProcessor extends EmailEventBase { }

```

## Resources


Read more about batch processing at these sites:

- [Smack My Batch Up : Batch Processing In Drupal 8](#) by Phil Norton July 2016
- Highly commented [source code for batch operations around line 561 for Drupal 10](#) (or search for 'batch operations')

Read more about the Queue API at these sites:

- Karim Boudjema from August 2018 has [some good examples using the queue API](#)
  - Sarthak TTN from Feb 2017 shows some [sample code on implementing cron and the queue API](#)
  - [There is a somewhat incomplete example](#) From Alan Saunders article on December 2021
- 
- 

[Back to top](#)

[Drupal at your fingertips](#) by [Selwyn Polit](#) is licensed under [CC BY 4.0](#) 

Page last modified: Jul 29 2023.

[Edit this page on GitHub](#)

This site uses [Just the Docs](#), a documentation theme for Jekyll.