**Queue API** https://api.drupal.org/api/drupal/modules!system!sy

stem.queue.inc/group/queue/7

Queue items to allow later processing.

The queue system allows placing items in a queue and processing them later. The system tries to ensure that only one consumer can process an item.

Before a queue can be used it needs to be created by **DrupalQueueInterface::createQueue().**

Items can be added to the queue by passing an arbitrary data object to **DrupalQueueInterface::createItem().**

**To process an item, call DrupalQueueInterface::claimItem() and specify how long you want to have a lease for working on that item. When finished processing, the item needs to be deleted by calling DrupalQueueInterface::deleteItem(). If the consumer dies, the item will be made available again by the DrupalQueueInterface implementation once the lease expires. Another consumer will then be able to receive it when calling DrupalQueueInterface::claimItem(). Due to this, the processing code should be aware that an item might be handed over for processing more than once.**

The $item object used by the DrupalQueueInterface can contain arbitrary metadata depending on the implementation. Systems using the interface should only rely on the data property which will contain the information passed to DrupalQueueInterface::createItem(). The full queue item returned by DrupalQueueInterface::claimItem() needs to be passed to DrupalQueueInterface::deleteItem() once processing is completed.

There are two kinds of queue backends available: **reliable, which preserves the order of messages and guarantees that every item will be executed at least once. The non-reliable kind only does a best effort to preserve order in messages and to execute them at least once but there is a small chance that some items get lost.** For example, some distributed back-ends like **Amazon SQS will be managing jobs for a large set of producers and consumers where a strict FIFO ordering will likely not be preserved. Another example would be an in-memory queue backend which might lose items if it crashes**. However, such a backend would be able to deal with significantly more writes than a reliable queue and for many tasks this is more important. See aggregator\_cron() for an example of how to effectively utilize a non-reliable queue. Another example is doing Twitter statistics -- the small possibility of losing a few items is insignificant next to power of the queue being able to keep up with writes. As described in the processing section, regardless of the queue being reliable or not, the processing code should be aware that an item might be handed over for processing more than once (because the processing code might time out before it finishes).

|  |  |
| --- | --- |
| [**DrupalQueue**](https://api.drupal.org/api/drupal/modules%21system%21system.queue.inc/class/DrupalQueue/7.x) | **Factory class for interacting with queues.** |
| [**MemoryQueue**](https://api.drupal.org/api/drupal/modules%21system%21system.queue.inc/class/MemoryQueue/7.x) | **Static queue implementation.** |
| [**SystemQueue**](https://api.drupal.org/api/drupal/modules%21system%21system.queue.inc/class/SystemQueue/7.x) | **Default queue implementation.** |

**Why Use Queues?**

Before getting into how to use and customise queues I thought it made sense to go into a few real world examples of how to use a queue to accomplish certain tasks.

* **Batch Processing**  
  If the module you are writing requires processing several items at once, which might not be from the same function call, then it might make sense to add them to a queue. These items can then be executed in one place. For example, let’s say that when you insert a node you want several other nodes to be created that act as inner pages. You would then set up a queue to add these pages in and also to add the menu items needed.
* **Sequential Processing Of Items**  
  When inserting taxonomy terms you will often want to insert certain terms before others if you want to maintain a parent-child relationship between terms. By using a queue you can make sure that all items are processed one after the other so that any terms that have children will be inserted first, before the child elements are created.
* **Delay Processing**  
  If you have large amounts of data to run through then doing so all at once might lead to the system overloading. So to spread out the load of these calculations you would add them to a queue and process them either slowly or when the system has the availability. An example of this might be when creating a payroll system or similar. You need to be sure that every calculation is complete so adding the items to be calculated into a queue means that the system load will be kept to a reasonable level. It also means that if the system does go down the calculations will still be in the queue ready to be processed.
* **Importing**  
  Importing content is a common task for many Drupal programmers, and sometimes putting together all the data needed for a node can be an expensive process. By figuring out what nodes need to be inserted and adding them to a queue you can then make sure that every node is inserted without overloading the system. I have been in the situation a couple of items where I have needed to insert large numbers of nodes and used a counting system to keep track of where the system was in the import process. With a queue I wouldn't have needed to write this code as I can be sure that all the items I add to it will be retrieved when I ask for them. In this way I can process as a subsection of the nodes without causing the system to fall over.
* **Preventing API Service Black Listing**  
  Many third party services (like Twitter) will keep track of the number of API requests that you do. If you exceed a certain limit they will simply block you from doing any more. If you have a large amount of items to process then it might be an idea to use a queue to spread out these items to prevent the API quota being exceeded.

### Reliable vs Non-reliable

Drupal 7 defines two types of queue class that you can use to do different things. These are **reliable** and **non-reliable**. Note that the term is non-reliable, not unreliable.

#### Reliable

With reliable queues you can be sure that every item will executed or processed at least once. A database table is used to store the queue, which means that the queue can exist over several requests. This also means that if the request fails then the queue remains intact.

The only way that Drupal knows that an object is reliable is that it extends the DrupalReliableQueueInterface interface. There is nothing inherently reliable about queue objects that means it is reliable.

This is the most commonly used type of queue and is the sytem default for new queue objects and has a wide variety of uses.

#### Non-reliable

Non-reliable queues are generally kept in memory only, so all items might exist in a single request. There is no guarantee that all items in the queue will be executed or will be executed in order. If the request fails then the queue might be lost.

An example of this sort of queue might be when sending data for processing to a cloud environment. When you send data over to a cloud environment there is usually little or no complex communication between servers, the data is just sent.

An example of this sort of queue might be when creating a Twitter service bot. There is generally little impact on the user experience if a single Tweet fails so creating a non-reliable queue to send a few Tweets out is an ideal solution.

### Drupal Queue Classes

There are three classes within the Queues API that provide the queue functionality. These are **DrupalQueue, SystemQueue and MemoryQueue**.

#### DrupalQueue

The DrupalQueue class consists of a single static method called get(). To use it you pass the name of the queue you want to retrieve and the method will return an object that you can use to interact with your queue. The following code is the bare minimum needed to get a queue object.

$my\_queue = DrupalQueue::get('my\_queue');

The queue object returned will be either a custom queue object or one of the default system objects, depending on what variables have been set. Any queue you create will have a unique set of items that will hang from the queue name that you issued.

You can send a second parameter to the get() method, which is a boolean value that forces the queue object to be reliable. The default value here is false, meaning that the get() method will not force the returned queue to be reliable.

For those of you familiar with object oriented programming you can view DrupalQueue as a factory class, since its only function is to create queue objects. The default returned object is SystemQueue.

#### SystemQueue

The SystemQueue class is the default queue class in Drupal. It is an example of a reliable queue class and so therefore implements the DrupalReliableQueueInterface interface. This class uses the database table **queue** to store and retrieve the queue. Any items that are retrieved from the queue are 'leased' to ensure that no two processes get the same queue item.

The queue table is created during the Drupal install process and has the following structure.

| **Column** | **Type** | **Comment** |
| --- | --- | --- |
| **item\_id** | int(10) unsigned Auto Increment | Primary Key: Unique item ID. |
| **name** | varchar(255) | The queue name. |
| **data** | longblob NULL | The arbitrary data for the item. |
| **expire** | int(11) | Timestamp when the claim lease expires on the item. |
| **created** | int(11) | Timestamp when the item was created. |

#### MemoryQueue

The MemoryQueue class is an example of a non-reliable queue class and therefore does not implement the DrupalReliableQueueInterface interface. All of the queue items are stored in memory as a $queue parameter in the class. This class also implements item leasing to stop the same items being retrieved from the queue over and over again.

### Using Drupal Queues

The createItem() method lets you add data to the queue and the numberOfItems() method lets you see how many items are present in the queue.

The MemoryQueue class will store items as they are, whereas the SystemQueue class will serialize the items before storing them in the database.

Example 1 - SystemQueue

*// Create queue object*

$queue = DrupalQueue::get('my\_queue');

*// Create item*

$item = [array](http://www.php.net/array)(

  'dataitem1' => 'something',

  'int' => 123

);

*// Add item to queue*

$queue­->createItem($item);

*// Report on number of items present*

echo $queue­->numberOfItems(); *// Prints "1"*

Example 2 - MemoryQueue

*// Set up the memory queue variable*

variable\_set('queue\_class\_memory', 'MemoryQueue');

*// Get the MemoryQueue object*

$queue = DrupalQueue::get('memory');

*// Set up the item*

$item = [array](http://www.php.net/array)(

  'dataitem1' => 'something',

  'int' => 123

);

*// Add the item to the queue*

$queue-­>createItem($item);

*// Report on number of items*

echo $queue­->numberOfItems(); *// Prints "1"*

To get an item from the queue we use the claimItem() method of the queue object. The return value of this method is an object containing a number of properties, the important one being the data property, which contains whatever we put into the queue. If no item is found in the queue then claimItem() will return FALSE. The claimItem() method takes an optional parameter of lease time. This is the amount of time (in seconds) that must pass after the item has been claimed before the item is put back into the queue. The default time for SystemQueue is 24 hours and the default time for MemoryQueue is 30 seconds. For example, to set the lease time to 100 seconds you would do the following $got\_item = $queue-­>claimItem(100);

*// Get the queue*

$queue = DrupalQueue::get('my\_queue');

*// Claim item from the queue*

$got\_item = $queue­->claimItem();

*// Print data from item*

echo $got\_item-­>data['dataitem1'];

**Example:** Load the latest version of the event node and pass it to a publishevent hook that we had created for that service. Here is that part of the code.

*/ Claim event item*

$event = $queue->claimItem();

if ($event !== **FALSE**) {

*// If we have an event then invoke publishevent hook in submodule*

  $result = module\_invoke($event->data['service'], 'publishevent', node\_load($event->data['nid']));

}

### Releasing And Deleting Items

Once we have our queue item, and have used it, we can then either release it (reset the lease time) or delete it. To release the item back into the queue use the **releaseItem()** method, passing the item object as the single parameter.

$queue­->releaseItem($got\_item);

To delete the item from the queue entirely use the deleteItem() method, passing the item object as the single parameter.

$queue­->deleteItem($got\_item);

# function hook\_cron\_queue\_info

Declare queues holding items that need to be run periodically.

While there can be only one [hook\_cron](https://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/hook_cron/7.x" \o "Perform periodic actions.)() process running at the same time, there can be any number of processes defined here running. Because of this, long running tasks are much better suited for this API. Items queued in [hook\_cron](https://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/hook_cron/7.x" \o "Perform periodic actions.)() might be processed in the same cron run if there are not many items in the queue, otherwise it might take several requests, which can be run in parallel.

# Return value

An associative array where the key is the queue name and the value is again an associative array. Possible keys are:

* **'worker callback'**: The name of an implementation of [callback\_queue\_worker](https://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/callback_queue_worker/7.x" \o "Work on a single queue item.)().
* **'time'**: (optional) How much time Drupal should spend on calling this worker in seconds. Defaults to 15.
* **'skip on cron'**: (optional) Set to TRUE to avoid being processed during cron runs (for example, if you want to control all queue execution manually).

### Code

function [**hook\_cron\_queue\_info**](https://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/hook_cron_queue_info/7.x)() {

$queues['aggregator\_feeds'] = array(

'worker callback' => '**[aggregator\_refresh](https://api.drupal.org/api/drupal/modules%21aggregator%21aggregator.module/function/aggregator_refresh/7.x" \o "Checks a news feed for new items.)**',

'time' => 60,

);

return $queues;

}