Taste the Power of Drupal Plugins!

Peter Wolanin
drupal.org/u/pwolanin

March 13, 2026
DrupalCamp NJ

Plugin Power Agenda

00. About Me

Core contributor and module maintainer since 2006

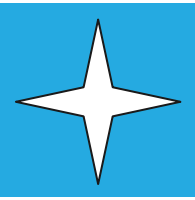
Drupal security team member since 2008

DrupalCamp NJ Organizer

One of the top 25 contributors to Drupal 8.0; helped create some of the core plugin types

Working for SciShield -> SciSure now, previously for Acquia.

01. Plugin Basics



- Encapsulate some reusable functionality inside a class that implements a specific interface
- Plugins of the same type can be thought of as different ways to satisfy the same need. For example, different ways to format a row in Views, or different remote providers for some kind of data
- Provided by modules to extend Drupal
- Often used when the user/developer can pick from among the different solutions to a need or problem
- OO design pattern: Plugin pattern

01. Plugin Basics

- Compared to older styles of finding implementations from different modules (like `hook_block()`), plugins are more flexible
- One module can provide many plugins of the same type, and one plugin can be used multiple times in different contexts
- As a last resort, you can even alter the plugin class being used in a specific case by taking advantage of the fact that you just need to match the interface

Spot the Plugins...

field.storage.node.field_image.yml

```yaml
langcode: en
status: true
id: node.field_image
field_name: field_image
entity_type: node
type: image
settings:
 target_type: file
 uri_scheme: public
module: image
```

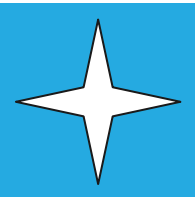
core/modules/image/src/Plugin/Field/FieldType/ImageItem.php

```php
/**
* Plugin implementation of the 'image' field.
*/
#[FieldType(
 id: "image",
 label: new TranslatableMarkup("Image"),
 category: "file_upload",
 default_widget: "image_image",
 default_formatter: "image",
 list_class: FileFieldItemList::class,
 constraints: [
   "ReferenceAccess" => [],
   "FileValidation" => []
  ],
)]
class ImageItem extends FileItem { … }
```

core/modules/file/src/Plugin/Validation/Constraint/FileValidationConstraint.php

```php
/**
* Validation File constraint.
*/
#[Constraint(
 id: 'FileValidation',
 label: new TranslatableMarkup('File Validation')
)]
class FileValidationConstraint
extends SymfonyConstraint { … }
```

02. Plugin Manager and IDs



- Every plugin type has a manager - registered as a service (available from the service container) and used to find and instantiate the desired plugin instance(s)
- Each plugin has an ID, which may be in its definition, or generated as a derivative (*WTF!*)
- For a given plugin ID a single class will be used for all plugin instances using that plugin ID
- A plugin instance is specified by the combination of plugin ID and its configuration values, potentially from a config entity (*WTF!*)

Plugin Manager - FactoryInterface

```php
/**
* Factory interface implemented by all plugin factories.
*/
interface FactoryInterface {

 /**
  * Creates a plugin instance based on the provided ID and configuration.
  *
  * @param string $plugin_id
  *   The ID of the plugin being instantiated.
  * @param array $configuration
  *   An array of configuration relevant to the plugin instance.
  *
  * @return object
  *   A fully configured plugin instance.
  *
  * @throws \Drupal\Component\Plugin\Exception\PluginException
  *   If the instance cannot be created, such as if the ID is invalid.
  */
 public function createInstance($plugin_id, array $configuration = []);
```

Simple Example: Instantiate Queue Worker

```php
/**
* The Drupal core Cron service.
*/
class Cron implements CronInterface {
 // ... //

/**
* The queue plugin manager.
*
* @var \Drupal\Core\Queue\QueueWorkerManagerInterface
*/
protected $queueManager;

/**
* Processes cron queues.
*/
protected function processQueues() {
 $max_wait = (float) $this->queueConfig['suspendMaximumWait'];

 // Build a stack of queues to work on.
 $queues = [];
 foreach ($this->queueManager->getDefinitions() as $queue_name => $queue_info) {
   if (!isset($queue_info['cron'])) {
     continue;
   }
   $queue = $this->queueFactory->get($queue_name);
   // Make sure every queue exists.
   $queue->createQueue();

   $worker = $this->queueManager->createInstance($queue_name);
```

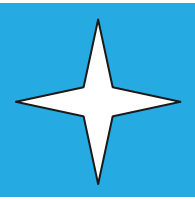
Simple Example: Queue Worker Interface

```php
/**
* Defines an interface for a QueueWorker.
*
* @see \Drupal\Core\Queue\QueueWorkerBase
* @see \Drupal\Core\Queue\QueueWorkerManager
* @see \Drupal\Core\Annotation\QueueWorker
* @see plugin_api
*/
interface QueueWorkerInterface extends PluginInspectionInterface {

 /**
  * Works on a single queue item.
  *
  * @param mixed $data
  *   The data that was passed to
  *   \Drupal\Core\Queue\QueueInterface::createItem() when the item was queued.
  *
  * @throws \Drupal\Core\Queue\RequeueException
  *   Processing is not yet finished.
  * @throws \Exception
  *   A QueueWorker plugin may throw an exception to indicate a problem.
  * @throws \Drupal\Core\Queue\SuspendQueueException
  *   Will not attempt to process further items from the current item's queue.
  * @throws \Drupal\Core\Queue\DelayedRequeueException
  *   To leave an item in the queue until its lock expires.
  *
  * @see \Drupal\Core\Cron::processQueues()
  */
 public function processItem($data);
```

Simple Example: Invoke Queue Worker

```php
/**
* The Drupal core Cron service.
*/
class Cron implements CronInterface {
 // ... //

/**
* Processes a cron queue.
*
* @param \Drupal\Core\Queue\QueueInterface $queue
*   The queue.
* @param \Drupal\Core\Queue\QueueWorkerInterface $worker
*   The queue worker.
*
* @throws \Drupal\Core\Queue\SuspendQueueException
*   If the queue was suspended.
*/
protected function processQueue(QueueInterface $queue, QueueWorkerInterface $worker) {
 $lease_time = $worker->getPluginDefinition()['cron']['time'];
 $end = $this->time->getCurrentTime() + $lease_time;
 while ($this->time->getCurrentTime() < $end && ($item = $queue->claimItem($lease_time))) {
   try {
     $worker->processItem($item->data);
     $queue->deleteItem($item);
   }
   catch (DelayedRequeueException $e) {
     // … //
     }
   }
```

03. Discovery



- Discovery == How the plugin manager finds its plugins
- Simple version - find all the PHP classes with the right attribute in the right namespace with the right interface
- The plugin **ID** is an attribute argument that must always be present
- The attribute arguments become part of the plugin **definition** - a fixed array of values used when creating a plugin instance
- Discovery is usually cached - your new plugin won't be found unless you do a cache rebuild

03. Discovery - Aside on PHP Attributes

- Since PHP 8.0 there has been support for attributes - metadata that can be associated with a class, method, etc.
- They use the format **`#[MyAttribute]`**
- Usually an attribute maps to a PHP class of the same name
- Arguments can be passed into attributes, and Drupal uses the named argument syntax when adding a plugin attribute

```
#[FieldType(
 id: "telephone",
 label: new TranslatableMarkup("Telephone number"),
 default_formatter: "basic_string"
)]
class TelephoneItem extends FieldItemBase { ... }
```

03. Discovery

Example of a plugin that automatically processes a queue during a Drupal cron - the attribute arguments provide the definition:

```php
namespace Drupal\media\Plugin\QueueWorker;

use // ... //

/**
 * Process a queue of media items to fetch their thumbnails.
 */
 #[QueueWorker(
  id: 'media_entity_thumbnail',
  title: new TranslatableMarkup('Thumbnail downloader'),
  cron: ['time' => 60]
 )]
 class ThumbnailDownloader extends QueueWorkerBase implements ContainerFactoryPluginInterface
 {  // ... // }
```

Queue Worker Definition Used in Processing

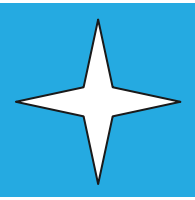
```php
/**
* The Drupal core Cron service.
*/
class Cron implements CronInterface {
 // ... //

/**
* Processes a cron queue.
*
* @param \Drupal\Core\Queue\QueueInterface $queue
*   The queue.
* @param \Drupal\Core\Queue\QueueWorkerInterface $worker
*   The queue worker.
*
* @throws \Drupal\Core\Queue\SuspendQueueException
*   If the queue was suspended.
*/
protected function processQueue(QueueInterface $queue, QueueWorkerInterface $worker) {
 $lease_time = $worker->getPluginDefinition()['cron']['time'];
 $end = $this->time->getCurrentTime() + $lease_time;
 while ($this->time->getCurrentTime() < $end && ($item = $queue->claimItem($lease_time))) {
   try {
     $worker->processItem($item->data);
     $queue->deleteItem($item);
   }
   catch (DelayedRequeueException $e) {
     // … //
     }
   }
```

Plugin Definition Versus Configuration

core/lib/Drupal/Component/Plugin/PluginBase.php

```
namespace Drupal\Component\Plugin;

/**
* Base class for plugins wishing to support metadata inspection.
*/
abstract class PluginBase implements PluginInspectionInterface, DerivativeInspectionInterface
{

// ... //

 /**
  * Constructs a \Drupal\Component\Plugin\PluginBase object.
  *
  * @param array $configuration
  *   A configuration array containing information about the plugin instance.
  * @param string $plugin_id
  *   The plugin ID for the plugin instance.
  * @param mixed $plugin_definition
  *   The plugin implementation definition.
  */
 public function __construct(array $configuration, $plugin_id, $plugin_definition) {
   $this->configuration = $configuration;
   $this->pluginId = $plugin_id;
   $this->pluginDefinition = $plugin_definition;
 }
```

03. Discovery



- Nuanced version - in addition to attribute + namespace + interface discovery, there is YAML file discovery and you may also see hook-based and other mechanisms
- Also, when you look, you'll see there is still BC support for annotation-based discovery, similar to PHP attribute discovery
- Discovery is also the process by which derivatives are generated (WTF #1, coming next)

YAML Discovery: Local Actions

menu_ui.links.action.yml

```yaml
entity.menu.add_link_form:
 route_name: entity.menu.add_link_form
 title: 'Add link'
 class: \Drupal\menu_ui\Plugin\Menu\LocalAction\MenuLinkAdd
 appears_on:
   - entity.menu.edit_form

entity.menu.add_form:
 route_name: entity.menu.add_form
 title: 'Add menu'
 appears_on:
   - entity.menu.collection
```

Local Action Plugin MenuLinkAdd

core/modules/menu_ui/src/Plugin/Menu/LocalAction/MenuLinkAdd.php

```php
namespace Drupal\menu_ui\Plugin\Menu\LocalAction;

use // ... //

/**
* Modifies the 'Add link' local action to add a destination.
*/
class MenuLinkAdd extends LocalActionDefault {

// ... //

 /**
  * {@inheritdoc}
  */
 public function getOptions(RouteMatchInterface $route_match) {
   $options = parent::getOptions($route_match);
   // Append the current path as destination to the query string.
   $options['query']['destination'] = $this->redirectDestination->get();
   return $options;
 }

}
```

Attributes vs. Annotations

Drupal 10 / 11

```php
/**
 * Process a queue of media items ...
 */
#[QueueWorker(
  id: 'media_entity_thumbnail',
  title: new TranslatableMarkup('Thumbnail
downloader'),
  cron: ['time' => 60]
)]
class ThumbnailDownloader extends
  QueueWorkerBase {}
```

Drupal 8 / 9

```php
/**
 * Process a queue of media items ...
 *
 * @QueueWorker(
 *   id = "media_entity_thumbnail",
 *   title = @Translation("Thumbnail
downloader"),
 *   cron = {"time" = 60}
 * )
 */
class ThumbnailDownloader extends
  QueueWorkerBase {}
```

04. Derivatives WTF: Discovery++

- A plugin definition can have an ID and reference a deriver class
- A deriver can generate many different final plugin IDs using this ID as a base ID, so it's an of enhancement on top of discovery

```
namespace Drupal\rest\Plugin\rest\resource;

use // ... //

#[RestResource(
 id: "entity",
 label: new TranslatableMarkup("Entity"),
 serialization_class: "Drupal\Core\Entity\Entity",
 deriver: EntityDeriver::class,
 uri_paths: [
   "canonical" => "/entity/{entity_type}/{entity}",
   "create" => "/entity/{entity_type}",
 ],
)]
class EntityResource extends ResourceBase implements DependentPluginInterface { ... }
```

04. Derivatives WTF: Discovery++

- A deriver implements

  `\Drupal\Component\Plugin\Derivative\DeriverInterface`
- Final ID of a derivative is typically `$base_id . ':' . $variant`
- Many of these derivers implement a plugin per entity type or per bundle of an entity type
- Some examples:
  - Field UI tabs (local tasks) to manage fields per entity bundle
  - Migration destination per entity type (ID like `entity:node`)
  - Entity reference selection (Same, like `default:comment`)
  - A block plugin for each Views block display

05. Config Entities WTF

- When I first learned about Drupal plugins I thought I had my head wrapped around it until this came up! I was lost again
- You might remember the plugin manager and its FactoryInterface where it took a 2nd parameter of configuration

```
interface FactoryInterface {

 /**
  * Creates a plugin instance based on the provided ID and configuration.
  *
  * @param string $plugin_id
  *   The ID of the plugin being instantiated.
  * @param array $configuration
  *   An array of configuration relevant to the plugin instance.
  *
  * @return object
  *   A fully configured plugin instance.
  */
 public function createInstance($plugin_id, array $configuration = []);
```

05. Config Entities WTF

- Remember that a plugin instance is the combination of the plugin ID and a specific set of configuration
- In most cases, if there is data passed in to the configuration parameter of the factory, it is from a config entity
- It turns out that some of the most common places you see plugins referenced is within the YAML exports of config entities such as field storage config entities, Views config entities, migrate plus config entities, and block config entities

05. Config Entities WTF

- A single config entity may have a collection with multiple plugin instances, each of which was created using a subset of the configuration entity's data
- The common pattern is that the config entity has a plugin collection, even when there is just one actual plugin
- The most important thing to remember today: it's much easier to load the config entity and get the plugin instance from it, rather than trying to directly create the the plugin instance

YAML Export of a Block Config Entity

block.block.olivero_powered.yml

```yaml
langcode: en
status: true
id: olivero_powered
theme: olivero
region: footer_bottom
weight: 0
provider: null
plugin: system_powered_by_block
settings:
 id: system_powered_by_block
 label: 'Powered by Drupal'
 label_display: '0'
 provider: system
visibility: {  }
```

```php
/** @var \Drupal\block\Entity\Block $entity */
$entity = \Drupal::entityTypeManager()->getStorage('block')
 ->load('olivero_powered');
$plugin = $entity->getPlugin();
```

Actual Block Plugin

core/modules/system/src/Plugin/Block/SystemPoweredByBlock.php

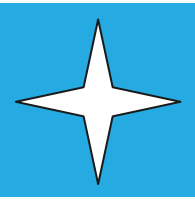
```php
namespace Drupal\system\Plugin\Block;

#[Block(
 id: "system_powered_by_block",
 admin_label: new TranslatableMarkup("Powered by Drupal")
)]
class SystemPoweredByBlock extends BlockBase {

 /**
  * {@inheritdoc}
  */
 public function defaultConfiguration() {
   return ['label_display' => FALSE];
 }

 /**
  * {@inheritdoc}
  */
 public function build() {
   return ['#markup' => '<span>' . $this->t('Powered by <a href=":poweredby">Drupal</a>',
      [':poweredby' => 'https://www.drupal.org']) . '</span>'];
 }

}
```

YAML Export of a Block Config Entity for a View

block.block.umami_views_block__articles_aside_block_1.yml

```yaml
langcode: en
status: true
id: umami_views_block__articles_aside_block_1
theme: umami
region: sidebar
weight: -7
provider: null
plugin: 'views_block:articles_aside-block_1'
settings:
 id: 'views_block:articles_aside-block_1'
 label: ''
 label_display: visible
 provider: views
 views_label: ''
 items_per_page: none
visibility:
 'entity_bundle:node':
   id: 'entity_bundle:node'
   negate: false
   context_mapping:
     node: '@node.node_route_context:node'
   bundles:
     article: article
```

06. Starting Places for You

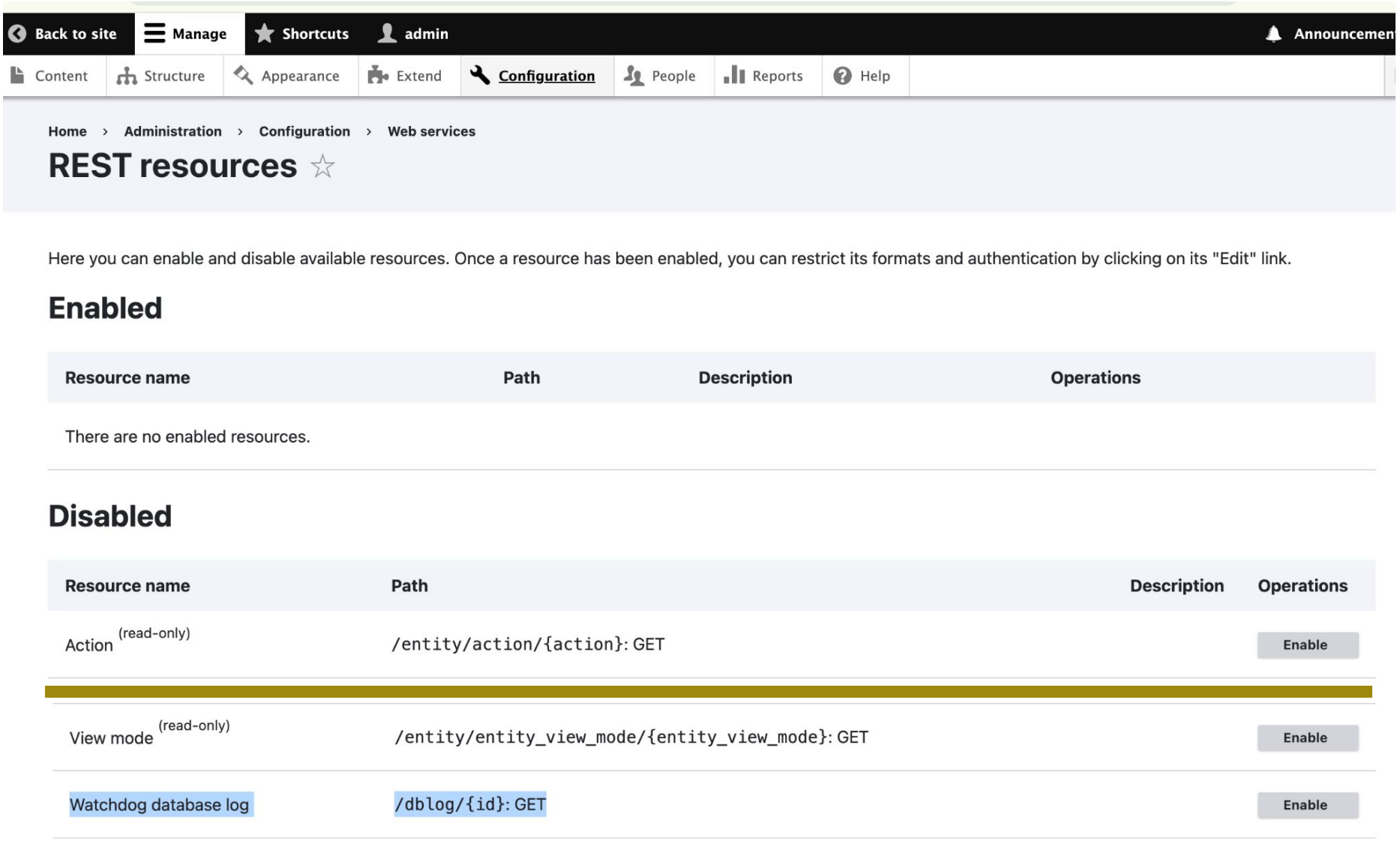


- Process a queue on cron (initially seen in 02)
- Define a simple block (just seen in 05)
- Implement a custom REST endpoint
  - Enable the core rest module and maybe get restui too
  - Any endpoint (plugin) needs a config entity to make it active (which restui can create)
  - DbLogResource example from core has a few flaws including lack of proper DI, but it gives us the basic idea

core/modules/dblog/src/Plugin/rest/resource/DbLogResource.php

```php
namespace Drupal\dblog\Plugin\rest\resource;
use // ... //

#[RestResource(
 id: "dblog",
 label: new TranslatableMarkup("Watchdog database log"),
 uri_paths: [
   "canonical" => "/dblog/{id}",
 ]
)]
class DbLogResource extends ResourceBase {

 /**
  * Responds to GET for a watchdog log entry with the specified ID.
  *
  * @param int $id
  *   The ID of the watchdog log entry.
  *
  * @return \Drupal\rest\ResourceResponse
  *   The response containing the log entry.
  */
 public function get($id = NULL) {
   if ($id) {
     $record = Database::getConnection()->select('watchdog', 'w')
       ->fields('w')->condition('wid', $id)->execute()->fetchAssoc();
     if (!empty($record)) {
       return new ResourceResponse($record);
     }
   }
   throw new NotFoundHttpException("Log entry with ID '$id' was not found");
 }
}
```

Create a Rest Resource Config Entity



YAML Export of a Rest Resource Config Entity

rest.resource.dblog.yml

```yaml
uuid: 2096e0a5-c829-4ac7-8e68-10ff64df8607
langcode: en
status: true
dependencies:
 module:
   - dblog
   - serialization
   - user
id: dblog
plugin_id: dblog
granularity: resource
configuration:
 methods:
   - GET
 formats:
   - json
   - xml
 authentication:
   - cookie
```

Rest Plugin Responses

127.0.0.1:8080/dblog/50?_format=json

Pretty-print


```
{
  "wid": "50",
  "uid": "1",
  "type": "user",
  "message": "Session opened for %name.",
  "variables": "a:1:{s:5:\"%name\";s:5:\"admin\";}",
  "severity": "6",
  "link": "",
  "location": "http://127.0.0.1:8080/user/login",
  "referer": "http://127.0.0.1:8080/user/login",
  "hostname": "127.0.0.1",
  "timestamp": "1744622270"
}
```


127.0.0.1:8080/dblog/50?_format=xml

This XML file does not appear to have any style information associated with it.


```
<response>
  <wid>50</wid>
  <uid>1</uid>
  <type>user</type>
  <message>Session opened for %name.</message>
  <variables>a:1:{s:5:"%name";s:5:"admin";}</variables>
  <severity>6</severity>
  <link/>
  <location>http://127.0.0.1:8080/user/login</location>
  <referer>http://127.0.0.1:8080/user/login</referer>
  <hostname>127.0.0.1</hostname>
  <timestamp>1744622270</timestamp>
</response>
```

06. Starting Places for You

Your plugin classes you should take advantage of

`\Drupal\Core\Plugin\ContainerFactoryPluginInterface` for DI

```php
namespace Drupal\media\Plugin\QueueWorker;

use // ... //

/**
 * Process a queue of media items to fetch their thumbnails.
 */
#[QueueWorker(
  id: 'media_entity_thumbnail',
  title: new TranslatableMarkup('Thumbnail downloader'),
  cron: ['time' => 60]
)]
class ThumbnailDownloader extends QueueWorkerBase implements ContainerFactoryPluginInterface
{  // ... // }
```

core/lib/Drupal/Core/Plugin/ContainerFactoryPluginInterface.php

```php
namespace Drupal\Core\Plugin;

use Symfony\Component\DependencyInjection\ContainerInterface;

/**
* Defines an interface for pulling plugin dependencies from the container.
*/
interface ContainerFactoryPluginInterface {

 /**
  * Creates an instance of the plugin.
  *
  * @param \Symfony\Component\DependencyInjection\ContainerInterface $container
  *   The container to pull out services used in the plugin.
  * @param array $configuration
  *   A configuration array containing information about the plugin instance.
  * @param string $plugin_id
  *   The plugin ID for the plugin instance.
  * @param mixed $plugin_definition
  *   The plugin implementation definition.
  *
  * @return static
  *   Returns an instance of this plugin.
  */
 public static function create(
   ContainerInterface $container,
   array $configuration,
   $plugin_id,
   $plugin_definition
  );
}
```

core/modules/media/src/Plugin/QueueWorker/ThumbnailDownloader.php

```php
namespace Drupal\media\Plugin\QueueWorker;
use // ... //

/**
* Process a queue of media items to fetch their thumbnails.
*/
#[QueueWorker(
 id: 'media_entity_thumbnail',
 title: new TranslatableMarkup('Thumbnail downloader'),
 cron: ['time' => 60]
)]
class ThumbnailDownloader extends QueueWorkerBase implements ContainerFactoryPluginInterface {

  // ... //

 /**
  * {@inheritdoc}
  */
 public static function create(ContainerInterface $container, array $configuration,
$plugin_id, $plugin_definition) {
   return new static(
     $configuration,
     $plugin_id,
     $plugin_definition,
     $container->get('entity_type.manager')
   );
 }
```

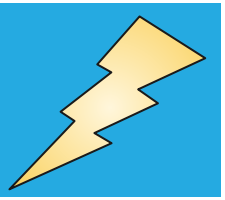
core/modules/media/src/Plugin/QueueWorker/ThumbnailDownloader.php

```php
#[QueueWorker(
 id: 'media_entity_thumbnail',
 title: new TranslatableMarkup('Thumbnail downloader'),
 cron: ['time' => 60]
)]
class ThumbnailDownloader extends QueueWorkerBase implements ContainerFactoryPluginInterface {

 protected EntityTypeManagerInterface $entityTypeManager;

 /**
  * @param array $configuration
  *   A configuration array containing information about the plugin instance.
  * @param string $plugin_id
  *   The plugin ID for the plugin instance.
  * @param mixed $plugin_definition
  *   The plugin implementation definition.
  * @param \Drupal\Core\Entity\EntityTypeManagerInterface $entity_type_manager
  *   Entity type manager service.
  */
 public function __construct(
   array $configuration,
   $plugin_id,
   $plugin_definition,
   EntityTypeManagerInterface $entity_type_manager
 ) {
   parent::__construct($configuration, $plugin_id, $plugin_definition);
   $this->entityTypeManager = $entity_type_manager;
 }
```

core/modules/media/src/Plugin/QueueWorker/ThumbnailDownloader.php

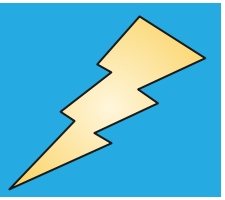
```php
namespace Drupal\media\Plugin\QueueWorker;
use // ... //

/**
* Process a queue of media items to fetch their thumbnails.
*/
#[QueueWorker(
 id: 'media_entity_thumbnail',
 title: new TranslatableMarkup('Thumbnail downloader'),
 cron: ['time' => 60]
)]
class ThumbnailDownloader extends QueueWorkerBase implements ContainerFactoryPluginInterface {

  // ... //

 /**
  * {@inheritdoc}
  */
 public function processItem($data) {
   /** @var \Drupal\media\Entity\Media $media */
   if ($media = $this->entityTypeManager->getStorage('media')->load($data['id'])) {
     $media->updateQueuedThumbnail();
     $media->save();
   }
 }

}
```

07. Define Your Own



- This is a more advanced use case, and will typically come up if you are authoring a complex contrib module or custom code for a web application (especially one codebase use for multiple sites)
- Some examples to think about:
  - Different ways to store or access secrets or credentials
  - Common interface to access different APIs with the same functionality (e.g. weather, translation, chemical SDS search, ...)
  - Different ways to render a row in a data table (if you are not using Views which has its own plugins for this)

07. Define Your Own



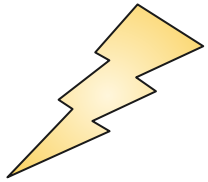
- An outline of what's needed:
  - New attribute class extending

    `\Drupal\Component\Plugin\Attribute\Plugin`
  - Namespace that your plugins will use
  - Plugin class interface
  - Plugin manager class
  - Service definition for the plugin manager
  - A plugin class implementation, at least in an automated test

Most of What's Needed is Seen in a Manager

```php
namespace Drupal\Core\Queue;
use // ... //

class QueueWorkerManager extends DefaultPluginManager {
 /**
  * @param \Traversable $namespaces
  *   An object that implements \Traversable which contains the root paths
  *   keyed by the corresponding namespace to look for plugin implementations.
  * @param \Drupal\Core\Cache\CacheBackendInterface $cache_backend
  *   Cache backend instance to use.
  * @param \Drupal\Core\Extension\ModuleHandlerInterface $module_handler
  *   The module handler.
  */
 public function __construct(
  \Traversable $namespaces,
  CacheBackendInterface $cache_backend,
  ModuleHandlerInterface $module_handler
 ) {
   parent::__construct(
     subdir: 'Plugin/QueueWorker',
     $namespaces,
     $module_handler,
     plugin_interface: 'Drupal\Core\Queue\QueueWorkerInterface',
     plugin_definition_attribute_name: \Drupal\Core\Queue\Attribute\QueueWorker::class
    );
   $this->setCacheBackend($cache_backend, 'queue_plugins');
   $this->alterInfo('queue_info');
 }
```

Some Key Points to Remember



START

Plugins of a type have a specific interface and a plugin instance is a specific combination of the plugin ID and configuration

NEXT

Plugins are provided by modules and typically discovered based on their namespace, interface, and a PHP attribute on the class

FINALLY

Most plugins that need configuration get it from a config entity, so load the relevant entity if you need to get its configured plugins

Further Reading / Resources

The official documentation:

- https://www.drupal.org/docs/drupal-apis/plugin-api

Misc posts (YMMV):

- https://drupalize.me/tutorial/what-are-plugins
- https://drupalize.me/blog/unraveling-drupal-8-plugin-system
- https://abh.ai/blog/design-patterns-drupal
- https://evolvingweb.com/blog/introduction-services-plugins-and-events-drupal

Thank you!

Questions?