Services and Dependency Injection

TABLE OF CONTENTS

Overview

Services provide a "decoupled" way to access classes and members. Services are pluggable and replaceable by registering them with a service container which makes them well suited to test with PHPUnit tests.

Services can be accessed using two possible methods: static and injected (using dependency injection).

Static

For `.module` files and classes which are not exposed to the service container (see below for a definition of the service container), you have to use the static method of retrieving the service:

```
// Access the Drupal JSON serialization service.
$this->jsonSerialization = \Drupal::service('serialization.json');
$this-jsonSerialization->decode($string);

// Access your custom service.
$this->salutationService = \Drupal::service('hello_world.salutation');
$this->salutationService->hello();
// Or.
$abc_retrieval_service = \Drupal::service('abc.aardvark_retrieval_service');
$aardvark_names = $abc_retrieval_service->getAardvarkNames();
```

You can also get the container and use that to get a service (and then use the service) e.g.:

```
$container = \Drupal::getContainer()
$this->keyValue = $container->get('keyvalue')->get($collection);
```

Static Shorthand methods

A few popular services also have shorthand methods in the core Drupal.php file for accessing them faster (and easier for IDE autocompletion), for example, `\Drupal::entityTypeManager()`. Check it out for services with shorthand methods:

e.g.

```
public static function routeMatch() {
  return static::getContainer()->get('current_route_match');
}

public static function currentUser() {
  return static::getContainer()->get('current_user');
}

public static function entityTypeManager() {
  return static::getContainer()->get('entity_type.manager');
}

public static function cache($bin = 'default') {
  return static::getContainer()->get('cache.' . $bin);
}
```

and many more..

Services in action

Most of these examples show Drupal configuration.

In this example we use the `config.factory` service (via the `::configFactory()` shortcut) to change the system email plugin to use our mail plugin:

```
/*
 *  \Drupal::configFactory() retrieves the configuration factory.
 *
 * This is mostly used to change the override settings on the configuration
 * factory. For example, changing the language, or turning all overrides on
 * or off.
 */

/**
 * Implements hook_install().
 */
function hello_world_install() {
  $config = \Drupal::configFactory()->getEditable('system.mail');
  $mail_plugins = $config->get('interface');
  if (in_array('hello_world', array_keys($mail_plugins))) {
    return; }
  $mail_plugins['hello_world'] = 'hello_world_mail';
  $config->set('interface', $mail_plugins);
  $config->save();
}
```

Here we use config.factory service to change some config variables:

```
$values = $form_state->getValues();
$address1 = $values['footer_address1'];
$address2 = $values['footer_address2'];

$config = \Drupal::configFactory()>getEditable('dat.header_footer_settings');
$config->set('footer_address1', $address1);
$config->set('footer_address2', $address2);
$config->save();
```

Here we use the config.factory service (via ::config() shorthand method) to load some values from Drupal config:

```
$config = \Drupal::config('dat.header_footer_settings');

$address1 = $config->get('footer_address1');
$address2 = $config->get('footer_address2');
$email = $config->get('footer_email');
$logo_url = $config->get('logo_url');
```

Get the email address for the site using the config.factory service:

```
// returns website@d9book.com
$to = \Drupal::configFactory()->getEditable('system.site')->get('mail');
```

ControllerBase shortcuts

ControllerBase.php comes prepackaged with functions to get the following services statically:

```
protected function entityTypeManager() {
protected function entityFormBuilder() {
protected function cache($bin = 'default') {
protected function config($name) {
protected function keyValue($collection) {
protected function state() {
protected function moduleHandler() {
protected function formBuilder() {
protected function currentUser() {
protected function languageManager() {
```

This allows quick access from within your controllers to these services if you need to do things like:

```
// Make an entityQuery
$storage = $this->entityTypeManager()->getStorage('node');
$query = $storage->getQuery();
$query
  ->accessCheck(TRUE)
  ->condition('type', 'article')
  ->condition('title', $name)
  ->count();
$count_nodes = $query->execute();

// Or.

// Get info about the current user.
$account = $this->currentUser();
$username = $account->getAccountName();
$uid = $account->id();
$message = "<br>Account info user id: " . $uid . " username: " . $username;
```

Injected/Dependency Injection

Using dependency injection is the preferred way to use services although it involves more steps. See the Dependency Injection section below for more details.

Using dependency injection requires that you create a constructor and a `create()` function in your controller class.

The create function gets the service container as a parameter and chooses the services it needs. The create function calls the constructor and passes the services as arguments and stores them as properties.

The process for a block (or plugin) is a little different:

Your block must implement ContainerFactoryPluginInterface. *Plugins only get access to the service container if they implement the `ContainerFactoryPluginInterface`* e.g.

```
class TestBlock extends BlockBase implements ContainerFactoryPluginInterface {
```

You must also add the extra parameters to the `create()` function and the constructor i.e. `$plugin_id` and `$plugin_definition` e.g.

```
public static function create(ContainerInterface $container, array $configuration, $plugin_id, $plugin_definition)
```

Here is an example of a block constructor with the `AccountProxyInterface` parameter added so we can inject that service:

```
public function __construct(array $configuration, $plugin_id, $plugin_definition, AccountProxyInterface $account) {
parent::__construct($configuration, $plugin_id, $plugin_definition);
$this->account = $account;
}
```

More about services and dependency injection at https://code.tutsplus.com/tutorials/drupal-8-properly-injecting-dependencies-using-di--cms-

26314

More about using dependency injection for blocks and other plugins https://chromatichq.com/blog/dependency-injection-drupal-8-plugins

Controller details

Here are the steps for implementing an injected service in a controller.

From: docroot/modules/custom/apitest/src/Controller/ApiTestController.php

1. Your controller must extend ControllerBase

```
class ApiTestController extends ControllerBase {
```

2. You need a protected variable to hold the service

```
/**
 *
 * The CmAPIClient.
 *
 * @var \Drupal\cm_api\CmAPIClient
 */
protected $cmAPIClient;
```

3. You need a `create()` function. This will get passed the `$container` so it can call it's `get()` member function to instantiate the service you need. This function then calls the constructor and passes it's parameters to it.

```
public static function create(ContainerInterface $container) {
  return new static(
    $container->get('cm_api.client')
  );
}
```

Note, you can pass multiple services by adding additional `$container->get()` calls like this:

```
public static function create(ContainerInterface $container) {
  return new static(
    $container->get('current_user'),
    $container->get('path.current'),
    $container->get('path.validator'),
  );
}
```

4. Your constructor will expect your newly instantiated service(s) as parameters:

```
/**
 * ApiTestController constructor.
 */
public function __construct(CmAPIClient $cmAPIClient) {
  $this->cmAPIClient = $cmAPIClient;
}
```

Similarly, if you are getting multiple services, add the additional parameters to the constructor, as well as assigning the variables. E.g.

```
public function __construct(AccountProxyInterface $account, CurrentPathStack $path_stack, PathValidatorInterface $path_validator) {
  $this->account = $account;
  $this->pathStack = $path_stack;
  $this->pathValidator = $path_validator;
}
```

5. In your code, use the protected variable (`$cmAPIClient`) to call functions in the service:

```
$result = $this->cmAPIClient->catchAll('POST', $body);
```

Rejoice! Note. No need to make any routing changes. Drupal handles all the parameters with the instructions provided. etc.

Controller Example 1

Here is a complete controller which uses the `current_user` service:

```
<?php

namespace Drupal\di_examples\Controller;

use Drupal\Core\Controller\ControllerBase;
use Drupal\Core\Session\AccountProxyInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class DiExamplesController extends ControllerBase {

  protected AccountProxyInterface $account;

  /**
   * Builds the response.
   */
  public function build() {

    $account = $this->account->getAccount();
    $username = $account->getAccountName();
    $uid = $account->id();

    $message = "<br>Account info user id: " . $uid . " username: " . $username;

    $build['content'] = [
      '#type' => 'item',
      '#markup' => $this->t($message),
    ];

    return $build;
  }

  public static function create(ContainerInterface $container) {
    return new static($container->get('current_user'));
  }

  public function __construct(AccountProxyInterface $account) {
    $this->account = $account;
  }

}
```

Controller Example 2

This controller uses 3 different services:

```
<?php

namespace Drupal\di_examples\Controller;

use Drupal\Core\Controller\ControllerBase;
```

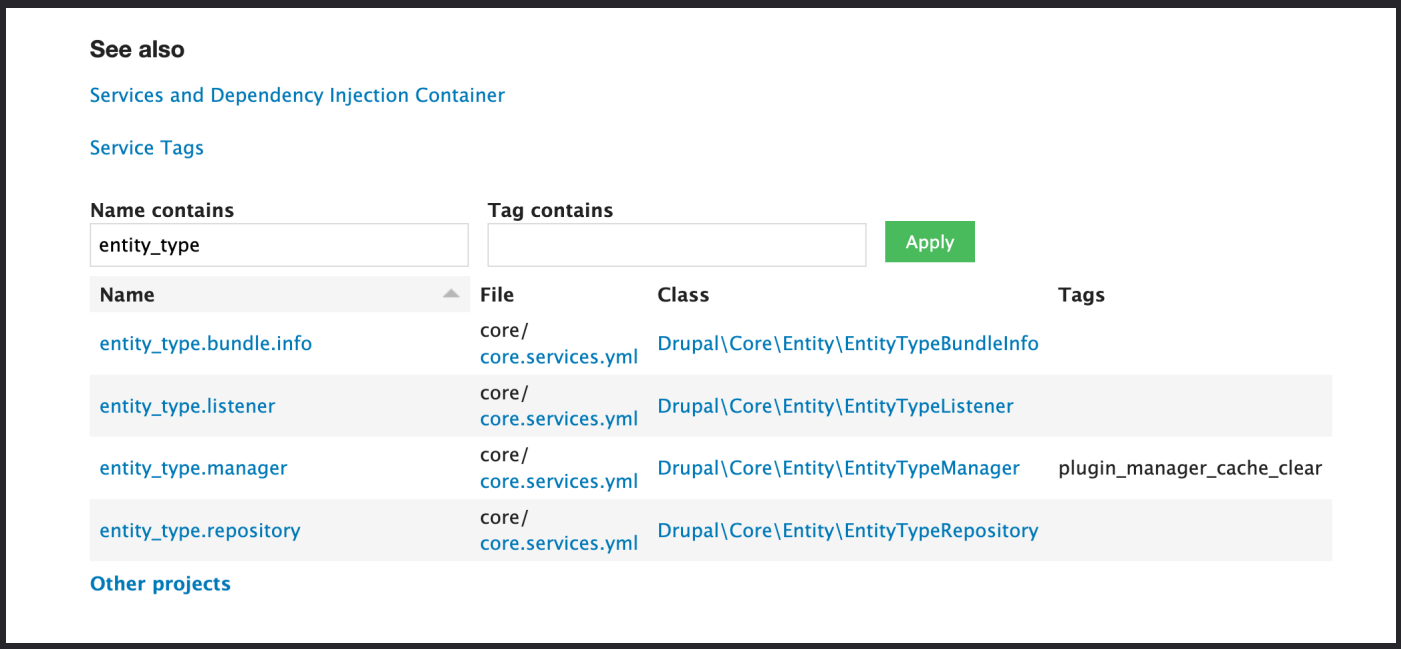
```php
use Drupal\Core\Path\CurrentPathStack;
use Drupal\Core\Path\PathValidatorInterface;
use Drupal\Core\Session\AccountProxyInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Returns responses for DI Examples routes.
 */
class DiExamplesController extends ControllerBase {

  protected AccountProxyInterface $account;
  protected CurrentPathStack $pathStack;
  protected PathValidatorInterface $pathValidator;

  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('current_user'),
      $container->get('path.current'),
      $container->get('path.validator'),
    );
  }

  public function __construct(AccountProxyInterface $account, CurrentPathStack $path_stack, PathValidatorInterface $path_validator) {
    $this->account = $account;
    $this->pathStack = $path_stack;
    $this->pathValidator = $path_validator;
  }


  /**
   * Builds the response.
   */
  public function build() {

    // Use the injected account.
    $account = $this->account->getAccount();

    // Use the ControllerBase static version.
    $account = $this->currentUser();

    $username = $account->getAccountName();
    $uid = $account->id();

    $message = "<br>Account info user id: " . $uid . " username: " . $username;

    $name = 'hello';

    // Use the ControllerBase static version to create an entityQuery.
    $storage = $this->entityTypeManager()->getStorage('node');
    $query = $storage->getQuery();
    $query
      ->condition('type', 'article')
      ->condition('title', $name)
      ->count();
    $count_nodes = $query->execute();
    $message .= "<br>Retrieved " . $count_nodes . " nodes";

    $path = $this->pathStack->getPath();
    $message .= "<br> Path: " . $path;
```

```
    $test_path = "/vote1";
    $valid_path = $this->pathValidator->isValid($test_path);
    $message .= "<br> Check for valid path: " . $test_path . " returned: " . $valid_path;



    $build['content'] = [
      '#type' => 'item',
      '#markup' => $this->t($message),
    ];

    return $build;
  }

}
```

Finding services

Here is the process to find a commonly used service, the entityTypeManager which is used for entityQueries.

You can look at https://api.drupal.org/api/drupal/services and search for entity_type. This will result in:



See also

Services and Dependency Injection Container

Service Tags

Name contains entity_type

Tag contains

Apply

Name	File	Class	Tags
entity_type.bundle.info	core/core.services.yml	Drupal\Core\Entity\EntityTypeBundleInfo	
entity_type.listener	core/core.services.yml	Drupal\Core\Entity\EntityTypeListener	
entity_type.manager	core/core.services.yml	Drupal\Core\Entity\EntityTypeManager	plugin_manager_cache_clear
entity_type.repository	core/core.services.yml	Drupal\Core\Entity\EntityTypeRepository	

Other projects

This tells you to that the service is in core.services.yml and that it is implemented in the EntityTypeManager class.

So in Drupal core's core.services.yml file you will find:

```
entity_type.manager:
  class: Drupal\Core\Entity\EntityTypeManager
  arguments: ['@container.namespaces', '@module_handler', '@cache.discovery', '@string_translation', '@class_resolver', '@entity.last_installed_schema.repository']
  parent: container.trait
  tags:
    - { name: plugin_manager_cache_clear }
```

Looking in Drupal.php, you will also find a shorthand method:

```
/**
 * Retrieves the entity type manager.
 *
 * @return \Drupal\Core\Entity\EntityTypeManagerInterface
 *   The entity type manager.
 */
public static function entityTypeManager() {
  return static::getContainer()->get('entity_type.manager');
}
```

So to use this statically, you can use the following:

```
$storage = \Drupal::entityTypeManager()->getStorage($entity_type);
$query = $storage->getQuery();
$query = \Drupal::entityQuery('node')
  ->accessCheck(TRUE)
  ->condition('type', 'page')
  ->condition('status', 1);
$nids = $query->execute();
```

To do this using dependency injection you will need to inject `entity_type.manager`. Follow the procedure outlined above in Controller details.

Creating a custom service

To make a service, you will need two parts: a `module.services.yml` file and a controller.

In the `module.services.yml` file. You need a machine name for the service and a class that implements it. E.g. in `kitchen_product.services.yml` you might have the following:

```
services:
  kitchen_product.product_manager_service:
    class: Drupal\kitchen_product\ProductManagerService
```

Arguments

You can specify optional arguments to pass to your service which might be needed when Drupal instantiates your service. Here we pass the `keyvalue` service as well as a Boolean value:

```
services:
  highway.road_generator:
    class: Drupal\highway\RoadGenerator
    arguments:
      - '@keyvalue'
      - "%highway.road.use_key_value_cache%"
```

The `%highway.road.user_key_value_cache%` value is a special argument which has been defined above the services key as a parameter. Its value will default to true on production:

```
parameters:
  highway.road.use_key_value_cache: true

services:
  dino_roar.roar_generator:
    class: Drupal\dino_roar\Jurassic\RoarGenerator
    arguments:
      - '@keyvalue'
      - "%highway.road.use_key_value_cache%"
```

This means that it is configurable. If you need to set it to false during development, you can easily override it on your local machine. Simply

add it to the development.services.yml file like this:

```
# Local development services.
#
# To activate this feature, follow the instructions at the top of the
# 'example.settings.local.php' file, which sits next to this file.
parameters:
  highway.road.use_key_value_cache: false
```

Note you can also pass strings in the form 'blah' surrounded by single quotes.

Passing the config factory to our service

As shown above, arguments use the "arguments" key, which can have an array of services, each preceeded by an @ symbol. Other values which are not services can also be passed. In the module.services.yml file below, we pass the config.factory service. You can find it in the core.services.yml file where you can see it maps to the Drupal\Core\Config\ConfigFactory class.

It looks like this in the core.services.yml:

```
config.factory:
  class: Drupal\Core\Config\ConfigFactory
  tags:
    - { name: event_subscriber }
    - { name: service_collector, tag: 'config.factory.override', call: addOverride }
  arguments: ['@config.storage', '@event_dispatcher', '@config.typed']
```

Here is how it looks in your module.services.yml:

```
services:
 hello_world.salutation:
   class: Drupal\hello_world\HelloWorldSalutation
   arguments: ['@config.factory']
```

Taxonomy Tree Custom Service

Here is an example where Daniel Sipos of https://www.webomelette.com creates a custom service to build a taxonomy tree. Read his article at https://www.webomelette.com/loading-taxonomy-terms-tree-drupal-8 . The repo is at https://github.com/upchuk/taxonomy_tree. His code is reproduced below:

Here is the taxonomy_tree.services.yml file:

```
services:
  taxonomy_tree.taxonomy_term_tree:
    class: Drupal\taxonomy_tree\TaxonomyTermTree
    arguments: ['@entity_type.manager']
```

And the controller:

```
<?php

namespace Drupal\taxonomy_tree;

use Drupal\Core\Entity\EntityTypeManager;

/**
 * Loads taxonomy terms in a tree
 */
class TaxonomyTermTree {

  /**
   * @var \Drupal\Core\Entity\EntityTypeManager
```

```
 */
protected $entityTypeManager;

/**
 * TaxonomyTermTree constructor.
 *
 * @param \Drupal\Core\Entity\EntityTypeManager $entityTypeManager
 */
public function __construct(EntityTypeManager $entityTypeManager) {
  $this->entityTypeManager = $entityTypeManager;
}

/**
 * Loads the tree of a vocabulary.
 *
 * @param string $vocabulary
 *   Machine name
 *
 * @return array
 */
public function load($vocabulary) {
  $terms = $this->entityTypeManager->getStorage('taxonomy_term')->loadTree($vocabulary);
  $tree = [];
  foreach ($terms as $tree_object) {
    $this->buildTree($tree, $tree_object, $vocabulary);
  }

  return $tree;
}

/**
 * Populates a tree array given a taxonomy term tree object.
 *
 * @param $tree
 * @param $object
 * @param $vocabulary
 */
protected function buildTree(&$tree, $object, $vocabulary) {
  if ($object->depth != 0) {
    return;
  }
  $tree[$object->tid] = $object;
  $tree[$object->tid]->children = [];
  $object_children = &$tree[$object->tid]->children;

  $children = $this->entityTypeManager->getStorage('taxonomy_term')->loadChildren($object->tid);
  if (!$children) {
    return;
  }

  $child_tree_objects = $this->entityTypeManager->getStorage('taxonomy_term')->loadTree($vocabulary, $object->tid);

  foreach ($children as $child) {
    foreach ($child_tree_objects as $child_tree_object) {
      if ($child_tree_object->tid == $child->id()) {
        $this->buildTree($object_children, $child_tree_object, $vocabulary);
      }
    }
  }
```

```
  }
}
```

Using your custom service

This is identical to using a Drupal built in service. These are the steps:

1. In your controller, make sure your controller extends `ControllerBase`.
2. Add protected variable in your class to hold your service.
3. Add a `create()` function to get the service(s) from the service container
4. Add a constructor which stores a link to each service so you can call functions in those services.

Note. Follow the slightly different steps for injecting services into blocks when using your service for blocks or plugins.

Dependency Injection

Overview

Dependency injection is the practice of "injecting" services. A service is any object managed by the Drupal Service container.

Drupal introduces the concept of services to decouple reusable functionality and makes these services pluggable and replaceable by registering them with a service container.

It is best practice to access any of the services provided by Drupal via the service container to ensure the decoupled nature of these systems is respected.

Services are used to perform operations like accessing the database or sending an e-mail. Rather than use PHP's native MySQL functions, we use the core-provided service via the service container to perform this operation so that our code can simply access the database without having to worry about whether the database is MySQL or SQLite, or if the mechanism for sending e-mail is SMTP or something else.

From https://www.drupal.org/docs/drupal-apis/services-and-dependency-injection/services-and-dependency-injection-in-drupal-8.

Service Container

The Service container is the PHP object which handles the instantiation of all required services. When you want to use a service, you ask the service container for one and then you can call methods on the service. The Drupal Service container is built on top of the Symfony Service container.

More at https://www.drupal.org/docs/drupal-apis/services-and-dependency-injection/services-and-dependency-injection-in-drupal-8.

For example, in `core.services.yml`, you will find the `email.validator` service which references the `EmailValidator` class. This is what you will see in `core.services.yml`:

```
email.validator:
  class: Drupal\Component\Utility\EmailValidator
```

Looking in the EmailValidator.php file, there is an `isValid()` function which you can call to validate email addresses. E.g.

```
$this->emailValidator->isValid()
```

Similarly, in `core.services.yml` there is a `current_route_match` service which references the class `CurrentRouteMatch`:

```
current_route_match:
  class: Drupal\Core\Routing\CurrentRouteMatch
  arguments: ['@request_stack']
```

Looking in `CurrentRouteMatch.php`, there is a `getRouteName()` function which can be used to get the current route name with a call like:

```
$this->currentRouteMatch->getRouteName()
```

Dig deeper in `core.services.yml` file for many more services.

Controller Example 1

Here is a complete controller which uses the current_user service:

```php
<?php

namespace Drupal\di_examples\Controller;

use Drupal\Core\Controller\ControllerBase;
use Drupal\Core\Session\AccountProxyInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class DiExamplesController extends ControllerBase {

  protected AccountProxyInterface $account;

  /**
   * Builds the response.
   */
  public function build() {

    $account = $this->account->getAccount();
    $username = $account->getAccountName();
    $uid = $account->id();

    $message = "<br>Account info user id: " . $uid . " username: " . $username;

    $build['content'] = [
      '#type' => 'item',
      '#markup' => $this->t($message),
    ];

    return $build;
  }

  public static function create(ContainerInterface $container) {
    return new static($container->get('current_user'));
  }

  public function __construct(AccountProxyInterface $account) {
    $this->account = $account;
  }

}
```

Controller Example 2

This controller uses 3 different services

```php
<?php

namespace Drupal\di_examples\Controller;

use Drupal\Core\Controller\ControllerBase;
use Drupal\Core\Path\CurrentPathStack;
use Drupal\Core\Path\PathValidatorInterface;
use Drupal\Core\Session\AccountProxyInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Returns responses for DI Examples routes.
```

```php
 * Returns responses for DI Examples routes.
 */
class DiExamplesController extends ControllerBase {

  protected AccountProxyInterface $account;
  protected CurrentPathStack $pathStack;
  protected PathValidatorInterface $pathValidator;

  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('current_user'),
      $container->get('path.current'),
      $container->get('path.validator'),
    );
  }

  public function __construct(AccountProxyInterface $account, CurrentPathStack $path_stack, PathValidatorInterface $path_validator) {
    $this->account = $account;
    $this->pathStack = $path_stack;
    $this->pathValidator = $path_validator;
  }


  /**
   * Builds the response.
   */
  public function build() {

    // Use the injected account.
    $account = $this->account->getAccount();

    // Use the ControllerBase static version.
    $account = $this->currentUser();

    $username = $account->getAccountName();
    $uid = $account->id();

    $message = "<br>Account info user id: " . $uid . " username: " . $username;

    $name = 'hello';

    // Use the ControllerBase static version to create an entityQuery.
    $storage = $this->entityTypeManager()->getStorage('node');
    $query = $storage->getQuery();
    $query
      ->accessCheck(TRUE)
      ->condition('type', 'article')
      ->condition('title', $name)
      ->count();
    $count_nodes = $query->execute();
    $message .= "<br>Retrieved " . $count_nodes . " nodes";

    $path = $this->pathStack->getPath();
    $message .= "<br> Path: " . $path;

    $test_path = "/vote1";
    $valid_path = $this->pathValidator->isValid($test_path);
    $message .= "<br> Check for valid path: " . $test_path . " returned: " . $valid_path;
```

```
    $build['content'] = [
      '#type' => 'item',
      '#markup' => $this->t($message),
    ];

    return $build;
  }

}
```

Blocks and other plugins

The process for a block (or plugin) is a little different:

Your block must implement `ContainerFactoryPluginInterface`. Plugins only get access to the service container if they implement the `ContainerFactoryPluginInterface` e.g.

```
class TestBlock extends BlockBase implements
ContainerFactoryPluginInterface {
```

You must also add the extra parameters to the `create()` and `__construct()` function i.e. `$plugin_id` and `$plugin_definition` e.g.

```
public static function create(ContainerInterface $container, array $configuration, $plugin_id, $plugin_definition)
```

Here is an example of a block constructor with the `AccountProxyInterface` parameter added as this is the service we want to inject:

```
public function __construct(array $configuration, $plugin_id, $plugin_definition, AccountProxyInterface $account) {
parent::__construct($configuration, $plugin_id, $plugin_definition);
$this->account = $account;
}
```

See more about using dependency injection for blocks and other plugins at https://chromatichq.com/blog/dependency-injection-drupal-8-plugins

Procedural to Class-based dependency injection

Here is the overview from the `Drupal.php` file for Drupal 9.5.0 of when and how to use dependency injection:

```
/**
 * Static Service Container wrapper.
 *
 * Generally, code in Drupal should accept its dependencies via either
 * constructor injection or setter method injection. However, there are cases,
 * particularly in legacy procedural code, where that is infeasible. This
 * class acts as a unified global accessor to arbitrary services within the
 * system in order to ease the transition from procedural code to injected OO
 * code.
 *
 * The container is built by the kernel and passed in to this class which stores
 * it statically. The container always contains the services from
 * \Drupal\Core\CoreServiceProvider, the service providers of enabled modules and any other
 * service providers defined in $GLOBALS['conf']['container_service_providers'].
 *
 * This class exists only to support legacy code that cannot be dependency
 * injected. If your code needs it, consider refactoring it to be object
 * oriented, if possible. When this is not possible, for instance in the case of
 * hook implementations, and your code is more than a few non-reusable lines, it
 * is recommended to instantiate an object implementing the actual logic.
 *
 * @code
```

```
 *   // Legacy procedural code.
 *   function hook_do_stuff() {
 *     $lock = lock()->acquire('stuff_lock');
 *     // ...
 *   }
 *
 *   // Correct procedural code.
 *   function hook_do_stuff() {
 *     $lock = \Drupal::lock()->acquire('stuff_lock');
 *     // ...
 *   }
 *
 *   // The preferred way: dependency injected code.
 *   function hook_do_stuff() {
 *     // Move the actual implementation to a class and instantiate it.
 *     $instance = new StuffDoingClass(\Drupal::lock());
 *     $instance->doStuff();
 *
 *     // Or, even better, rely on the service container to avoid hard coding a
 *     // specific interface implementation, so that the actual logic can be
 *     // swapped. This might not always make sense, but in general it is a good
 *     // practice.
 *     \Drupal::service('stuff.doing')->doStuff();
 *   }
 *
 *   interface StuffDoingInterface {
 *     public function doStuff();
 *   }
 *
 *   class StuffDoingClass implements StuffDoingInterface {
 *     protected $lockBackend;
 *
 *     public function __construct(LockBackendInterface $lock_backend) {
 *       $this->lockBackend = $lock_backend;
 *     }
 *
 *     public function doStuff() {
 *       $lock = $this->lockBackend->acquire('stuff_lock');
 *       // ...
 *     }
 *   }
 * @endcode
 *
 * @see \Drupal\Core\DrupalKernel
 */
```

Drush services commands

List all services

drush devel:services or drush dcs or

```
drush eval "print_r(\Drupal::getContainer()->getServiceIds());"
```

drush dcs | grep "PART OF SERVICE NAME" can find a service e.g. drush dcs | grep "access" will find all services with access in the name.

```
$ drush devel:services
- access_arguments_resolver_factory
- access_check.contact_personal
- access_check.cron
- access_check.csrf
- access_check.custom
- access_check.db_update
- access_check.default
- access_check.entity
- access_check.entity_bundles
- access_check.entity_create
- access_check.entity_create_any
- access_check.entity_delete_multiple
- access_check.field_ui.form_mode
- access_check.field_ui.view_mode
- access_check.header.csrf
- access_check.node.add
- access_check.node.preview
- access_check.node.revision
- access_check.permission
- access_check.quickedit.entity_field
- access_check.theme
- access_check.update.manager_access
- access_check.user.login_status
- access_check.user.register
- access_check.user.role
- access_manager
- account_switcher
- admin_toolbar_tools.helper
- ajax_response.attachments_processor
- ajax_response.subscriber
...
```

Generate custom service

Drush provides a great starting point by generating some useful code that you can easily build on. Consider using this facility as you write your code.

From https://www.drush.org/latest/generators/service_custom

`drush generate service:custom`. Generates a custom Drupal service

Also there are these gems:

`drush generate service:logger`. Generates a logger service

`drush generate service:breadcrumb-builder`. Generates a breadcrumb builder service

`drush generate service:event-subscriber`. Generates an event subscriber

`drush generate service:middleware`. Generates a middleware

`drush generate service:param-converter`. Generates a param converter service

`drush generate service:path-processor`. Generates a path processor service

`drush generate service:request-policy`. Generates a request policy service

`drush generate service:response-policy`. Generates a response policy service

`drush generate service:route-subscriber`. Generates a route subscriber

etc.

Resources

- Services and Dependency Injection in the Drupal.org API documentation
- Drupal 8: Properly injecting dependencies using DI by Danny Sipos from May 2016
- Dependency injection in Drupal 8 plugins (or blocks) by Märt Matoo from March 2017
- Drupal 8: Properly Injecting Dependencies Using DI by Danny Sipos May 2016
- Inject a service in Drupal 8 showing an example of injecting http_client (Guzzle) by J M Olivas July 2015

Back to top

Page last modified: Jul 23 2023.

Edit this page on GitHub