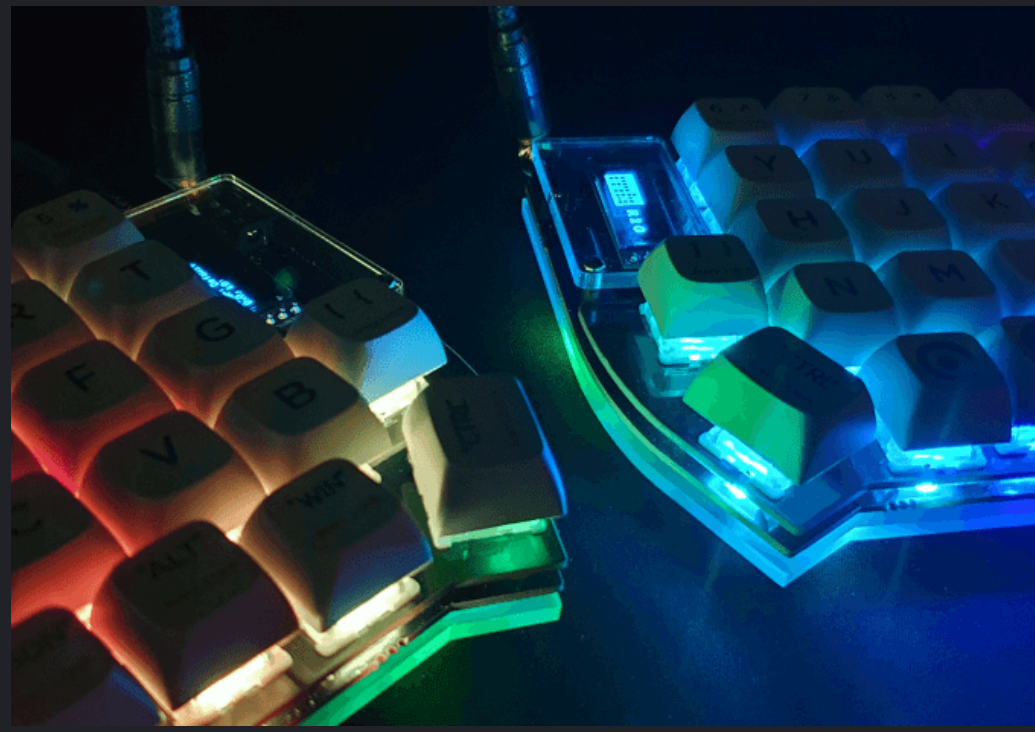
An Introduction To Drupal Services

Philip Norton

DrupalCamp England 2025

Philip Norton

- Developer at Code Enigma
- Writer at `#! code` (www.hashbangcode.com)
- NWDUG co-organiser



Source Code

- Talk is available at: https://github.com/hashbangcode/drupal-services-talk
- All code seen can be found at: https://github.com/hashbangcode/drupal_services_example
- I have also written about Drupal services on www.hashbangcode.com

An Introduction To Drupal Services

What Is A Service?

- Used in all parts of Drupal and many modules.
- Built on the **Symfony Service Container** system.
- A service describes an object in Drupal.
- Dependency injection is used to inject services into other services.
- Forms, controllers, and plugins have services built in.
- Simple to use and powerful.

```php
\Drupal::service('thing');
```

What Services Exist?

- Lots!

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

- Prints a list of over 600 services.
- Most are in the form `date.formatter`.
- Some are in the form `Drupal\Core\Datetime\DateFormatterInterface`, and are used in autoloading.

Using A Service

Using A Service

- Grab the service.
- Use it.

```
$pathManager = \Drupal::service('path_alias.manager');
$normalPath = $pathManager->getPathByAlias('somepath');
```

Using A Service

- You can also chain the call.

```
$normalPath = \Drupal::service('path_alias.manager')->getPathByAlias('somepath');
```

Using A Service

- However! Most of the time you don't want to be using `\Drupal::service()`.
- Drupal will inject the services you need into your service.
- This is called **dependency injection**.

Dependecny Injection

A quick introduction.

Dependency Injection

Dependecy injection sounds complicated, but its just the practice of **injecting the things the object needs**, instead of **baking them into the class**.

Dependency Injection

- Let's say you had this class (not a Drupal thing).

```
class Page {
  protected $database;
  public function __construct() {
    $this->database = new PDO('mysql:host=localhost;dbname=test', 'username', 'password');
  }
}
$page = new Page();
```

- What happens if you want to change the credentials? Or change the database itself?
- You would need to edit the class.

Dependency Injection

- We can change this to inject the database dependency as we create the Page object.

```
class Page {
  protected $database;
  public function __construct(DbConnectionInterface $database) {
    $this->database = $database;
  }
}
$database = new MysqlDatabase();
$page = new Page($database);
```

Dependency Injection

- Drupal handles all of the object creation for us and will create services with all of the required objects in place.
- All we need to do is ask for our service.

Why Use Dependency Injection In Drupal?

Let's try to create the `path_alias.manager` service to translate a path *without* using Drupal's dependency injection system.

The `path_alias.manager` service wraps the `\Drupal\path_alias\AliasManager` class.

```
use Drupal\path_alias\AliasManager;

$aliasManager = new AliasManager($pathAliasRepository, $pathPrefixes, $languageManager, $cache, $time);
```

We need to fill in the missing dependencies of `$pathAliasRepository`, `$pathPrefixes`, `$languageManager`, `$cache`, and `$time`.

Let's start with `$pathAliasRepository`.

The `$pathAliasRepository` property is an instance of `\Drupal\path_alias\AliasRepository`.

```
use Drupal\path_alias\AliasManager;
use Drupal\path_alias\AliasRepository;

$pathAliasRepository = new AliasRepository($connection);

$aliasManager = new AliasManager($pathAliasRepository, $pathPrefixes, $languageManager, $cache, $time);
```

The `AliasRepository` class takes a property of `$connection`.

The `$connection` property is a connection to the database, which we can create using the `\Drupal\Core\Database\Database` class.

```
use Drupal\path_alias\AliasManager;
use Drupal\path_alias\AliasRepository;
use Drupal\Core\Database\Database;

$connection = Database::getConnection();
$pathAliasRepository = new AliasRepository($connection);

$aliasManager = new AliasManager($pathAliasRepository, $pathPrefixes, $languageManager, $cache, $time);
```

Next, let's look at `$pathPrefixes`.

The `$pathPrefixes` property is an instance of `AliasPrefixList`, which has more dependencies.

```php
use Drupal\path_alias\AliasManager;
use Drupal\path_alias\AliasPrefixList;
use Drupal\path_alias\AliasRepository;
use Drupal\Core\Database\Database;

$connection = Database::getConnection();
$pathAliasRepository = new AliasRepository($connection);

$pathPrefixes = new AliasPrefixList($cid, $cache, $lock, $state, $alias_repository);

$aliasManager = new AliasManager($pathAliasRepository, $pathPrefixes, $languageManager, $cache, $time);
```

The `$cid` property of `AliasPrefixList` is easy as that's just a string.

```
use Drupal\path_alias\AliasManager;
use Drupal\path_alias\AliasPrefixList;
use Drupal\path_alias\AliasRepository;
use Drupal\Core\Database\Database;

$connection = Database::getConnection();
$pathAliasRepository = new AliasRepository($connection);

$cid = 'path_alias_whitelist';
$pathPrefixes = new AliasPrefixList($cid, $cache, $lock, $state, $alias_repository);

$aliasManager = new AliasManager($pathAliasRepository, $pathPrefixes, $languageManager, $cache, $time);
```

The `$cache` property is an object of type `\Drupal\Core\Cache\CacheFactoryInterface`, so we can use `\Drupal\Core\Cache\CacheFactory` to create this. We first need to create a `\Drupal\Core\Site\Settings` object to create that.

```
use Drupal\path_alias\AliasManager;
use Drupal\path_alias\AliasPrefixList;
use Drupal\path_alias\AliasRepository;
use Drupal\Core\Database\Database;
use Drupal\Core\Site\Settings;

$connection = Database::getConnection();
$pathAliasRepository = new AliasRepository($connection);

$cid = 'path_alias_whitelist';
$settings = Settings::getInstance();
$default_bin_backends = ['bootstrap' => 'cache.backend.chainedfast'];
$cacheFactory = new CacheFactory($settings, $default_bin_backends);
$cache = $cacheFactory->get('bootstrap');
$pathPrefixes = new AliasPrefixList($cid, $cache, $lock, $state, $alias_repository);

$aliasManager = new AliasManager($pathAliasRepository, $pathPrefixes, $languageManager, $cache, $time);
```

Anyone else lost?

```
$pathManager = \Drupal::service('path_alias.manager');
$normalPath = $pathManager->getPathByAlias('somepath');
```

Seems easier, right?

Creating Your Own Services

Creating Services

- All services are defined in a `[module].services.yml` file in your module directory.

```
services:
  services_simple_example.simple_service:
    class: \Drupal\services_simple_example\SimpleService
```

Creating Services

- Create the class for your service.

```php
<?php

namespace Drupal\services_simple_example;

class SimpleService implements SimpleServiceInterface {

  public function getArray():array {
    return [];
  }
}
```

Creating Services

- You can now use your service like any Drupal service.

```
$simpleService = \Drupal::service('services_simple_example.simple_service');
$array = $simpleService->getArray();
```

Creating Services - Arguments

- Your services can accept a number of arguments.
  - `@` for another service (`@config.factory`).
  - `%` for a parameter (`%site.path%`).
  - 'config' = A string, in this case ‘config’.

Creating Services - Arguments

- Most commonly, we want to inject our service dependencies.

```
services:
  services_argument_example.single_argument:
    class: \Drupal\services_argument_example\SingleArgument
    arguments: ['@serialization.json']
```

Creating Services - Arguments

- Our service class needs to accept the arguments.

```php
<?php

namespace Drupal\services_argument_example;

use Drupal\Component\Serialization\SerializationInterface;

class SingleArgument implements SingleArgumentInterface {

  public function __construct(protected SerializationInterface $serializer) {}
}
```

Creating Services - Arguments

- The service can be used within the class.

```
public function removeItemFromPayload(string $payload, int $id):string {
  $payload = $this->serializer->decode($payload);
  foreach ($payload as $key => $item) {
    if ($item['id'] === $id) {
      unset($payload[$key]);
    }
  }
  return $this->serializer->encode($payload);
}
```

Creating Services - Autowiring

- You don't need to add all of your dependencies by hand, you can use autowiring to do this for you.
- Autowiring works by nominating services that correspond to interfaces.

```
services:
  Drupal\Component\Serialization\SerializationInterface: '@serialization.json'
```

Creating Services - Autowiring

- Then, we need to add the `autowire: true` directive to the service definition for our service.

```
services:
  services_autowire_example.autowire_example:
    class: \Drupal\services_autowire_example\AutowireExample
    autowire: true
```

Creating Services - Autowiring

- Alternatively, you can set a default in your service file that all services will be autowired.

```
services:
  _defaults:
    autowire: true

  services_autowire_example.autowire_example:
    class: \Drupal\services_autowire_example\AutowireExample
```

Creating Services - Autowiring

- Create your class as normal. The interfaces you nominate will be translated into services and automatically injected into your constructor.

```php
<?php

namespace Drupal\services_autowire_example;

use Drupal\Component\Serialization\SerializationInterface;

class AutowireExample implements AutowireExampleInterface {

  public function __construct(protected SerializationInterface $serializer) {
  }
}
```

Controllers And Forms

Controllers And Forms

- Some types of Drupal object (especially Controllers and Forms) don't use *.services.yml files.
- Instead they implement `\Drupal\Core\DependencyInjection\ContainerInjectionInterface`.
- Drupal will see this and use a method called `create()` to create the service.
- The `create()` method must return an instance of the service object.

Controllers And Forms

- Best practice is to assign the properties you need in the `create()` method.

```
class ControllerExample extends ControllerBase {

  protected $dateFormatter;

  public static function create(ContainerInterface $container) {
    $instance = new static();
    $instance->dateFormatter = $container->get('date.formatter');
    return $instance;
  }

  /// ..
}
```

Plugins

- Plugins have a similar interface called `\Drupal\Core\Plugin\ContainerFactoryPluginInterface`
- This has the same `create()` method system, although you need to pass the plugin arguments to the parent controller.

Tips For Creating Services

- Don't use `\Drupal::sercices()` inside your service classes, use depedency injection instead.
- Use **SOLID** principles. Create small service classes that perform one task.
- Keep constructors as simple as possible. Just assign your dependencies to properties.
- Don't "hand off" dependencies to internal classes, use additional services.

Tips For Creating Services

- Services make unit testing much easier.
  - Test your services on their own with unit testing.
  - Then move up to functional testing for test services working together.
  - Functional tests can test your module controllers, forms, drush commands etc.

Altering Services

Altering Services

- All services can be modified to change their funcitonality.
- This can be done in two ways, depending on your needs.
  - Decorating
  - Altering

Altering Services: Decorating

- Services can be decorated to create your own serive that extends another service.
- The original service will still exist, but you will have a new service that accepts the same arguments.

```
services:
  services_decorator_example.decorated_json:
    class: \Drupal\services_decorator_example\DecoratedJson
    decorates: serialization.json
```

Altering Services: Altering

- Override the serivce completely and replace it with your own.
- Create a class that has the name `[ModuleName]ServiceProvider`, which extends the class `\Drupal\Core\DependencyInjection\ServiceProviderBase`.
- Drupal will pick up this class and run the `register()` and `alter()` methods.

Altering Services: Altering

- The `alter()` method is can be used to alter an existing service.

```php
<?php

namespace Drupal\joke_api_stub;

use Drupal\Core\DependencyInjection\ContainerBuilder;
use Drupal\Core\DependencyInjection\ServiceProviderBase;

class JokeApiStubServiceProvider extends ServiceProviderBase {
  public function alter(ContainerBuilder $container) {
    // Replace the \Drupal\joke_api\JokeApi class with our own stub class.
    $definition = $container->getDefinition('joke_api.joke');
    $definition->setClass('Drupal\joke_api_stub\JokeApiStub');
  }
}
```

Demo!

Next Steps

There's much more to Drupal services, try looking up

- `autoconfigure: true`
- Tagged services
- Access control
- Logging
- Event handlers

Resources

- Services and DI on `#! code`
  https://www.hashbangcode.com/tag/dependency-injection
- Custom code seen is code available at
  https://github.com/hashbangcode/drupal_services_example
- Services and Dependency Injection -
  https://www.drupalatyourfingertips.com/services
- Structure of a service file
  https://www.drupal.org/docs/drupal-apis/services-and-dependency-injection/structure-of-a-service-file

Questions?

- Slides: https://github.com/hashbangcode/drupal-services-talk

Thanks!

- Slides: https://github.com/hashbangcode/drupal-services-talk

Philip Norton hashbangcode.com fosstodon.org@hashbangcode
fosstodon.org@philipnorton42