How to register layouts with Layout Plugin

For Drupal 8.2.x

https://www.drupal.org/node/2578731

**The**[**Layout plugin**](https://www.drupal.org/project/layout_plugin)**module allows modules or themes to register layouts, and for other modules (like**[**Panels**](https://www.drupal.org/project/panels)**or**[**Display Suite**](https://www.drupal.org/project/ds)**) to use them.**

The Layout plugin API is also included [experimentally](https://www.drupal.org/core/experimental) in Drupal 8.3.x core. This documentation will be updated for the core API as it stabilizes. See the [change record on the Layout plugin](https://www.drupal.org/node/2821240) for more information. See here for info about transitioning from [Layout plugin to layout discovery experimental module](https://www.drupal.org/node/2619128) in Drupal 8.3.x.

There are several ways to provide a layout. We'll discuss each in the sections below, starting with the simplest, most common case and building up to some of the more advanced techniques!

### Table of contents

1. [Registering layouts using \*.layouts.yml](https://www.drupal.org/node/2578731#using-layouts-yml)
   * [The simplest case](https://www.drupal.org/node/2578731#simplest-case)
   * [Registering your own template and using 'theme'](https://www.drupal.org/node/2578731#using-theme-key)
   * [Registering your own library and using 'library'](https://www.drupal.org/node/2578731#using-library-key)
   * [Using an alternate 'class'](https://www.drupal.org/node/2578731#using-class-key)
2. [Registering a layout as a PHP class with annotations](https://www.drupal.org/node/2578731#using-php-annotations)
3. [Registering dynamic layouts using derivatives](https://www.drupal.org/node/2578731#using-derivatives)
4. [Full annotation reference](https://www.drupal.org/node/2578731#full-annotation-reference)

## **Registering layouts using \*.layouts.yml**

The most basic way to register layouts, is to put a \*.layouts.yml file in your module or theme.

The first part of the file name is the machine name of your module or theme, so if for example, your module's machine name is my\_custom\_module, you'd call the file my\_custom\_module.layouts.yml.

This file should be placed in the top-level directory of your module or theme, and you'll need to rebuild the cache (for example, with Drush it's drush cr) for your changes to the file to be picked up.

In the next few sections, we'll cover the different keys and values you can use in this file.

### The simplest case

Note: Some info here was true for layout plugin contrib module, but some things changed when the layout plugin system moved into core 8.3, see [#2872957: Document that the 'css' property no longer works after upgrading from layout plugin](https://www.drupal.org/project/drupal/issues/2872957).

Here's an example of a super simple \*.layouts.yml:

one\_column:

label: One column

category: My Layouts

template: templates/one-column

regions:

main:

label: Main content

two\_column:

label: Two column

category: My Layouts

template: templates/two-column

regions:

main:

label: Main content

sidebar:

label: Sidebar

This registers two layouts: "One column" and "Two column". The 'label' and 'category' keys are required!

You can see that the first layout declares a single "Main content" region, and the second has both a "Main content" and "Sidebar" region.

At minimum, each layout should specify a Twig template. ([Versions before Drupal 8.3 expected a 'css' key also. This is now deprecated](https://www.drupal.org/node/2872957)) The 'template' should not include the .html.twigextension on the file, which means if you specified templates/one-column the actual file will be templates/one-column.html.twig.

(Note: You DON'T need to write your own *hook\_theme()* or declare any libraries - layout\_plugin takes care of this for you.)

In your Twig template, you can use tokens like {{ content.main }} and {{ content.sidebar }}to output the content of each region. So, for example, templates/two-column.html.twig could look like:

<div class="two-column">

<div class="main-region">

{{ content.main }}

</div>

<div class="sidebar-region">

{{ content.sidebar }}

</div>

</div>

**This is the simplest, and easiest way to register a new layout! For most use cases, you shouldn't need to read any further.**

But if you want to do something more advanced, please read on...

### Registering your own template and using 'theme'

In the first section, we simply gave layout\_plugin the 'template' key and it automatically registered it with the theme system.

It's also possible to tell layout\_plugin to use a particular theme hook (which you've manually registered using hook\_theme()). This can be useful if, for example, you want to use the same template to render multiple layouts.

To do this, you need to first register your template using [hook\_theme()](https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/function/hook_theme/8). For example, if wanted to declare a theme hook called "advanced\_layout\_1" you put this in your \*.module (if a module) or \*.theme (if a theme) file:

// Replace "MY\_MODULE\_OR\_THEME" with the machine name of your module or theme.

function MY\_MODULE\_OR\_THEME\_theme() {

return [

'advanced\_layout\_1' => [

'template' => 'templates/advanced-layout-1',

// layout\_plugin expects the theme hook to be declared with this:

'render element' => 'content',

],

];

}

Then in your \*.layouts.yml file, don't use 'template', but specify the machine name of your theme hook in 'theme'.

For example, if your theme hook is "advanced\_layout\_1", your \*.layouts.yml could look like:

advanced\_layout\_1:

label: Advanced Layout 1

category: My Layouts

theme: advanced\_layout\_1

regions:

main:

label: Main content

(Note: You can also give a specific theme suggestion in 'theme', like "advanced\_layout\_1\_\_alternate\_suggestion", by putting the suggestion after a double underscore "\_\_".)

### Registering your own library and using 'library' to add CSS or JS

It's possible to tell layout\_plugin to use a particular library (which you've manually registered using \*.libraries.yml). This can be useful if, for example, you want to use the same library to render multiple layouts, need to specify dependencies on other libraries, or include Javascript as well as CSS.

To do this, you need to first register your library with the [asset management system](https://www.drupal.org/developing/api/8/assets) using \*.libraries.yml. Here's a simple example:

advanced-layout-library:

version: 1.x

css:

theme:

css/advanced-layout-library.css: {}

js:

js/advanced-layout-library.js: {}

dependencies:

- core/jquery

Then in your \*.layouts.yml file, specify the machine name of your library in 'library'.

For example, if your library is "advanced-layout-library", your \*.layouts.yml could look like:

advanced\_layout\_2:

label: Advanced Layout 2

category: My Layouts

template: templates/advanced-layout-2

# Replace "MY\_MODULE\_OR\_THEME" with the machine name of your module or theme.

library: MY\_MODULE\_OR\_THEME/advanced-layout-library

regions:

main:

label: Main content

### Using an alternate 'class'

So far, in the previous examples we haven't had to write much (or any) PHP code! This works by using the default layout class (\Drupal\layout\_plugin\Plugin\Layout\LayoutDefault) for any layouts that don't specify a 'class' key.

However, it's possible to use a custom 'class' for a layout! This can be useful if, for example, you want your layout to have custom settings so that users can change the way the layout is rendered.

(Note: You can specify a custom 'class' with layouts registered in both modules and themes, BUT the class itself MUST be in a module. Themes can't contain autoloadable class files.)

Here's an example layout class which provides a settings form:

/\*\*

\* @file

\* Contains \Drupal\my\_custom\_module\Plugin\Layout\MyLayoutClass

\*/

namespace Drupal\my\_custom\_module\Plugin\Layout;

use Drupal\Core\Form\FormStateInterface;

use Drupal\Core\StringTranslation;

use Drupal\layout\_plugin\Plugin\Layout\LayoutBase;

class MyLayoutClass extends LayoutBase {

use StringTranslationTrait;

/\*\*

\* {@inheritdoc}

\*/

public function defaultConfiguration() {

return parent::defaultConfiguration() + [

'extra\_classes' => 'Default',

];

}

/\*\*

\* {@inheritdoc}

\*/

public function buildConfigurationForm(array $form, FormStateInterface $form\_state) {

$configuration = $this->getConfiguration();

$form['extra\_classes'] = [

'#type' => 'textfield',

'#title' => $this->t('Extra classes'),

'#default\_value' => $configuration['extra\_classes'],

];

return $form;

}

/\*\*

\* @inheritDoc

\*/

public function submitConfigurationForm(array &$form, FormStateInterface $form\_state) {

parent::submitConfigurationForm($form, $form\_state);

$this->configuration['extra\_classes'] = $form\_state->getValue('extra\_classes');

}

}

Then your \*.layouts.yml could look like:

advanced\_layout\_3:

label: Advanced Layout 3

category: My Layouts

class: '\Drupal\my\_custom\_module\Plugin\Layout\MyLayoutClass'

template: templates/advanced-layout-3

library: my\_custom\_module/advanced-layout

regions:

main:

label: Main content

And in your Twig template, you now have access to the {{ settings.extra\_classes }}variable. So, your templates/advanced-layout-3.html.twig could look like:

<div class="my-advanced-layout {{ settings.extra\_classes }}">

<div class="main-region">

{{ content.main }}

</div>

</div>

## **Registering a layout as a PHP class with annotations**

In all the examples above, we've been using the \*.layouts.yml file to register layouts. That is the easiest way to declare a layout if don't need to give an alternate layout class.

While it's possible to specify an alternate layout 'class' in the \*.layouts.yml file, if you have a one-off layout that uses a custom class, it can be easier register it in PHP with annotations! (Or if your layouts are derivatives, it's the only way, which we'll discuss in more detail in the next section.)

To do this, you put a @Layout annotation above the layout class using the same keys as in the \*.layouts.yml. Here's a simple example:

/\*\*

\* A very advanced custom layout.

\*

\* @Layout(

\* id = "advanced\_layout\_4",

\* label = @Translation("Advanced Layout 4"),

\* category = @Translation("My Layouts"),

\* template = "templates/advanced-layout-4",

\* regions = {

\* "main" = {

\* "label" = @Translation("Main content"),

\* }

\* }

\* )

\*/

class AdvancedLayout4 extends LayoutBase {

// Override any methods you'd like to customize here!

}

(Note: declaring layouts this way can ONLY be done from a module. Themes cannot include plugins declared using PHP classes and annotations.)

## **Registering dynamic layouts using derivatives**

So far, we've been registering single layout plugins, one at a time. However, "derivatives" are sets of plugins that are registered dynamically based on some other data.

For example, if you wanted to create a flexible layout builder, you'd give the site administrator some user interface to declare a new layout, and the configuration for each layout would be saved as a config entity. Then you'd register a layout class that used a "deriver" to dynamically register a layout plugin for each of those saved config entities.

So, first you'd register the layout class, for example:

/\*\*

\* @file

\* Contains \Drupal\my\_custom\_module\Plugin\Layout\FlexibleLayout

\*/

namespace Drupal\my\_custom\_module\Plugin\Layout;

use Drupal\layout\_plugin\Plugin\Layout\LayoutBase;

/\*\*

\* A layout from our flexible layout builder.

\*

\* @Layout(

\* id = "flexible\_layout",

\* deriver = "Drupal\my\_custom\_module\Plugin\Deriver\FlexibleLayoutDeriver"

\* )

\*/

class FlexibleLayout extends LayoutBase {

/\*\*

\* {@inheritdoc}

\*/

public function build(array $regions) {

$render\_array = parent::build($regions);

// Since this is a flexible layout builder, you probably need to do

// something special to render the layout, so we override the ::build()

// method which is responsible for creating a render array.

return $render\_array;

}

}

Notice how the annotation is incomplete (i.e., it's missing some of the keys necessary to register a layout) and includes a 'deriver' key. It's the deriver's job to iterate over the data about the layouts and fill in the rest of the information about the layout.

Also, when declaring layouts in this way, unless you can point to a specific 'template' or 'theme', you'll probably need to override the layout's ::build() method to provide custom code for rendering the layout.

Now, here is what an example deriver could look like:

/\*\*

\* @file

\* Contains \Drupal\my\_custom\_module\Plugin\Deriver\FlexibleLayoutDeriver

\*/

namespace Drupal\my\_custom\_module\Plugin\Deriver;

use Drupal\Component\Plugin\Derivative\DeriverBase;

/\*\*

\* Makes a flexible layout for each layout config entity.

\*/

class FlexibleLayoutDeriver extends DeriverBase {

/\*\*

\* {@inheritdoc}

\*/

public function getDerivativeDefinitions($base\_plugin\_definition) {

// Here we need to magically get a list of the config entities.

// I leave this as an exercise for the reader. :-)

$config\_entities = [];

// Now we loop over them and declare the derivatives.

foreach ($config\_entities as $entity) {

$derivative = $base\_plugin\_definition;

// Here we fill in any missing keys on the layout annotation.

$derivative['label'] = $entity->label();

$derivative['category'] = $entity->getCategory();

$derivative['regions'] = $entity->getRegions();

$this->derivatives[$entity->id()] = $derivative;

}

return $this->derivatives;

}

}

(Note: declaring layouts this way can ONLY be done from a module. Themes cannot include plugins declared using PHP classes and annotations.)

## **Full annotation reference**

Each layout definition **must** have the following keys:

**label**

The human-readable name of the layout.

**category**

The human-readable category to which the layout belongs.

**regions**

Array of regions in the layout. The keys are the regions' machine names and the  
values are sub-arrays containing the following elements:

* label (required): The human-readable name of the region.

**theme**

If specified, the theme hook which will be used to render this layout. It's  
expected that the module or theme which registers the layout will also register this  
theme hook. If you use this key, you cannot use template.

**template**

If specified, the template to use to render the layout, relative to the given  
path, without the .html.twig extension. If given, the template will be  
automagically registered with the theme system. If you use this key, you cannot use  
theme.

Each layout definition can also have the following **optional** keys:

**type**

Defines how the layout will be used in a page. Can be one of:

* full: The layout is used for an entire page, not just the  
  main content.
* page (default): The layout is used for just the main page response  
  (i.e., whatever is returned by the controller).
* partial: A partial layout that can be used for sub-regions --  
  roughly analogous to Mini Panels in Drupal 7.

**description**

Optional description of the layout.

**path**

Base path of all assets (templates, CSS, icon, etc.), relative to the current  
module.

**library**

The asset library to load for this layout. If given, it's expected that the module  
or theme which registers the layout has also registered the library in its  
\*.libraries.yml file. [Click here for more  
information about asset libraries in Drupal 8.](https://www.drupal.org/developing/api/8/assets)

**css**

If given, a CSS file to attach when rendering this layout, relative to the given

path. If you use this key, you cannot use library.

**icon**

The screenshot/preview icon for this layout, relative to the given  
path.

# How to render layouts using Layout Plugin

For 8.3.x