Drupal 8

Module Development

Lakshmi Narasimhan

Drupal 8 module development

Learn how to write Drupal 8 modules

Lakshmi Narasimhan

This book is for sale at http://leanpub.com/drupal8book

This version was published on 2017-06-12

Tweet This Book!

Please help Lakshmi Narasimhan by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Drupal 8 module development by @lakshminp via @leanpub

The suggested hashtag for this book is #drupal8book.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#drupal8book

Contents

Basic Concepts

Drupal 8 adopts a host of modern PHP practices, thus enhancing the developer experience. For someone coming from Drupal 7, many of these concepts might sound new. The aim of this chapter is to provide familiarity with these practices.

Composer

Composer is PHP equivalent of Python's pip, Ruby's bundler and Node's npm.

Why composer?

The average PHP developer wrote 3.5 frameworks in their lifetime. This was before the era of package management. Every time I had to implement user authentication in my project, I had a collection of scripts which I reused and customized. Then came along PEAR. PEAR did the job, but had some flaws. To point out one, PEAR installs packages globally instead of installing them on a per-project basis. So, if you install Acme libary v1.2, you had to use Acme v1.2 throughout all of your projects. Thankfully, Composer has succeeded PEAR as the de facto package manager for PHP. It helps manage projects and dependencies on a per project basis.

Composer solves the following problems:

Managing project dependencies

If you are working on a non trivial project, there is a very good chance you are using another external library or module. Composer manages all these packages and their dependencies automatically.

Automatic loading of PHP packages in your project

Autoloading is a mechanism which allows us to use PHP classes from other files without the tedium of writing a long list of `require()` or `include()` lines in our PHP file. Composer does this complex autoloading for you.

Updating your project

Composer provides a standard way to update your project or its dependencies.

The Drupal context

Modules and themes in Drupal are managed by drush, specifically drush make. Both composer and drush make do the same thing, i.e. download and install specific versions of Drupal modules and themes. Composer has an extra ingredient called autoload. This automagically figures out where your dependency files are there and includes them in your code as needed. Drupal 8 core adopted composer as the dependency manager. So if composer is great, why isn't everybody using it? To bridge this gap, Drupal 8 brought in a module called composer manager. This module helps manage composer dependencies of contrib modules. This might otherwise be done by editing Drupal 8 core's root composer.json and adding our module's composer dependencies there(which amounts to hacking the core).

So, if module acme depends on package foo, it will have a composer.json in the module's root directory with foo added as a dependency. Composer manager will add foo to Drupal's vendor directory. Starting with Drupal 8.1, we no longer need Composer manager to manage module dependencies. It can be done by composer alone. Let's take a look at how we pull this off.

Make sure you have the latest Drupal 8 installed and setup. You can install composer for your operating system using the instructions in the Composer website[1]. Let's download the address module[2] using composer.

```
$ composer require drupal/address
  [InvalidArgumentException]                                          \

  Could not find package drupal/address at any version for your minimum-stabilit\
y (dev). Check the package spelling or your minimum-stability

require [--dev] [--prefer-source] [--prefer-dist] [--no-progress] [--no-update] \
[--no-scripts] [--update-no-dev] [--update-with-dependencies] [--ignore-platform\
-reqs] [--prefer-stable] [--prefer-lowest] [--sort-packages] [-o|--optimize-auto\
loader] [-a|--classmap-authoritative] [--] [<packages>]...
```

Not much luck. This is because our address module is not present in the list of places where composer typically searches for packages. Let's add the repository where all Drupal modules and themes are present.

```
$ composer config repositories.drupal composer https://packages.drupal.org/8
```

this will append the following config to our composer.json.

[1] https://getcomposer.org/
[2] https://www.drupal.org/project/address

```
"repositories": {
    "drupal": {
        "type": "composer",
        "url": "https://packages.drupal.org/8"
    }
}
```

What this does is to tell composer to search for a given package at `https://packages.drupal.org/8` in addition to its default list of places. Why can't all Drupal modules be moved to packagist? Why create another repository? Given that most of Drupal related artifacts are managed through Drupal.org, moving all of them to the default composer packagist[3] would involve a lot of effort. Hence, Drupal.org provides its own repository of composer metadata for Drupal projects.

NOTE that this is a one time thing per project and you need not add the drupal repository every time you want to add a module through composer.

At the time of writing this, you can check the on the progress of Drupal+Composer usage in this ticket[4].

Now, another shot at installing the address module.

```
$ composer require drupal/address
Using version 1.x-dev for drupal/address
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
.
.
.
Writing lock file
Generating autoload files
> Drupal\Core\Composer\Composer::preAutoloadDump
> Drupal\Core\Composer\Composer::ensureHtaccess
```

Here's where composer shines! It automatically installs the dependencies of the address module itself and autoloads them. Our address module's `composer.json` looks like this:

[3] https://packagist.org/

[4] https://www.drupal.org/node/2551607

```
{
  "name": "drupal/address",
  "type": "drupal-module",
  "description": "Provides functionality for storing, validating and displaying \
international postal addresses.",
  "homepage": "http://drupal.org/project/address",
  "license": "GPL-2.0+",
  "require": {
    "commerceguys/intl": "~0.7",
    "commerceguys/addressing": "~1.0",
    "commerceguys/zone": "~1.0"
  },
  "minimum-stability": "dev"
}
```

where `commerceguys/*` are its dependencies. These get installed inside the top-level vendor directory.

```
~/n/d/vendor ❯ ls
asm89/        bin/          doctrine/  fabpot/      jcalderonzumba/  paragonie/      phpunit/    stack/       twig/       zendframework/
autoload.php  commerceguys/ easyrdf/   guzzlehttp/  masterminds/     phpdocumentor/  psr/        symfony/     web.config
behat/        composer/     egulias/   ircmaxell/   mikey179/        phpspec/        sebastian/  symfony-cmf/ wikimedia/
~/n/d/vendor ❯ cd commerceguys/
~/n/d/v/commerceguys ❯ pwd
/home/lakshmi/n1/drupal-8.2.6/vendor/commerceguys
~/n/d/v/commerceguys ❯ ls
addressing/  enum/  intl/  zone/
~/n/d/v/commerceguys ❯
```

Address module dependencies in vendor directory

We can even go a notch higher and manage Drupal fully using composer.

```
$ composer create-project drupal-composer/drupal-project:8.x-dev <my_drupal_top_\
level_dir> --stability dev --no-interaction
```

Namespaces

You will end up using a lot of code written by other developers in your modules. There is a good chance that you will use the same class name declared in another library you use. A hacky way to avoid this is to use a verbose naming convention, like `MyModule_Common_Test_Parser`.

Namespaces solve this problem by grouping a set of related classes, interfaces, functions and constants under a common name. This is similar to having files by the same name under different directories. Again, like how directories can have subdirectories, namespaces can also have a similar hierarchy of sub-namespaces.

The following code declares a namespace called `Ticket`.

```
namespace Ticket;
```

This line is usually placed at the beginning of the file. It is a convention to have just one namespace per file. All the identifiers belonging to that namespaces can be written following the namespace declaration.

```
<?php
namespace Ticket;

class Task {

  // whatever

}
```

Let's add a few other ticket types in different files under the `Ticket` namespace.

```
<?php
namespace Ticket;

class Bug {}


<?php
namespace Ticket;

class Story {}
```

Let's say that our project is BugZap, so we create a top level namespace called `BugZap` and `Ticket` as a sub namespace. Our new code will look like this:

```
// Task.php
<?php
namespace BugZap\Ticket;

class  Task {}

// Bug.php
<?php
namespace BugZap\Ticket;
```

```
class  Bug {}

// Story.php
<?php
namespace BugZap\Ticket;

class  Story {}
```

To use the above class

Namespaces are case insensitive.

Annotations

Annotations are PHP comments which hold metadata about your function or class. They do not directly affect program semantics as they are comment blocks. They are read and parsed at runtime by an annotation engine.

Annotations are already used in other PHP projects for various purposes. Symfony2 uses annotations for specifying routing rules. Doctrine uses them for adding ORM related metadata.Though handy in various situations, their utility is debated about a lot, like:

1. How to differentiate between annotations and actual user comments?
2. Why put business logic inside comment blocks. Shouldn't they be a part of core language semantics?
3. Annotations blur the boundary between code and comments. If the developer misses an annotation(Remember, its not a program semantic). It might compile fine, but might not work as expected. Even worse, annotations are hard to test and debug.

```
    Most of the acquisitions are around the concept of annotations implemented as co\
    mments in PHP. There is however, a [[https://wiki.php.net/rfc/annotations][propo\
    sal]] to add it as a first class language feature. In the meantime, we are stuck\
     to using comment based annotations.
```

Annotations are not all evil. They make it easier to inject behaviour without adding a lot of boilerplate. Here's an example taken from stackoverflow[5] which shows how annotations can cut a lot of boilerplate code.

Let's say we want to inject `weapon` object to a `Soldier` instance.

[5] http://stackoverflow.com/questions/3623499/how-is-annotation-useful-in-php

```
class Weapon {
    public function shoot() {
        print "... shooting ...";
    }
}

class Soldier {
    private $weapon;

    public function setWeapon($weapon) {
        $this->weapon = $weapon;
    }

    public function fight() {
        $this->weapon->shoot();
    }
}
```

If this dependency injection is done manually, then:

```
$weapon = new Weapon();

$soldier = new Soldier();
$soldier->setWeapon($weapon);
$soldier->fight();
```

We could go a step further and decouple the DI and put it in an external file, like:

Soldier.php

```
$soldier = Container::getInstance('Soldier');
$soldier->fight(); // ! weapon is already injected
```

soldierconfig.xml

```
<class name="Soldier">
    <!-- call setWeapon, inject new Weapon instance -->
    <call method="setWeapon">
        <argument name="Weapon" />
    </call>
</class>
```

If we use annotations instead:

```
class Soldier {
    ...

    /**
     * @inject $weapon Weapon
     */
    public function setWeapon($weapon) {
        $this->weapon = $weapon;
    }
```

Annotations also give the additional advantage of having both code and metadata co-located. I think this is another reason why it was decided[6] to use annotations and do away with external configuration files in Drupal 8(at least for plugins).

The Drupal part

Drupal 8 borrows annotations syntax from Doctrine[7]. Drupal 7 had metadata tucked away in info hooks. This involves reading the whole module file into memory for every request. Annotations, on the other hand, are tokenized and parsed and don't incur as much memory requirements as in the hook based approach. Also, docblocks are cached by the opcode cache.

Syntax

Drupal annotations are nested key value pairs very similar to json dumps. There are some gotchas though. You MUST use double quotes for strings and no quotes at all for numbers. Lists are represented by curly brackets and booleans by TRUE and FALSE without quotes. Here's a code dump of annotations in action, from `TextDefaultFormatter.php` file in `text` module.

[6] http://www.drupal4hu.com/node/377

[7] http://docs.doctrine-project.org/projects/doctrine-common/en/latest/reference/annotations.html

```
/**
 * Plugin implementation of the 'text_default' formatter.
 *
 * @FieldFormatter(
 *   id = "text_default",
 *   label = @Translation("Default"),
 *   field_types = {
 *     "text",
 *     "text_long",
 *     "text_with_summary",
 *   },
 *   quickedit = {
 *     "editor" = "plain_text"
 *   }
 * )
 */
class TextDefaultFormatter extends FormatterBase {
...
```

1. Routing and Controllers

Routes

The routing system of Drupal 8 is a complete rewrite of Drupal 7 and its previous versions' `hook_menu`. A Drupal route is a URL path which returns content. This returned content is usually specified as a method of a controller class.

The routing system is powered by Symfony's HTTP Kernel component, which we will revisit later. It is not necessary to understand the Symfony Kernel component in order to work with routes.

Your first route

Let's dive straight away and create a new route. First, we shall create a module to hold all the code in this chapter. You can checkout the code:

```
$ git clone git@github.com:drupal8book/myroute.git
$ cd myroute
$ git checkout -f simple-route
```

or walk along with me by typing the code or using drupalconsole[1] to generate it.

```
$ drupal generate:module

 Enter the new module name:
 > myroute

 Enter the module machine name [myroute]:
 >

 Enter the module Path [/modules/custom]:
 >

 Enter module description [My Awesome Module]:
 > Routing and controllers
```

[1] https://drupalconsole.com/

```
 Enter package name [Custom]:
 > D8MD

 Enter Drupal Core version [8.x]:
 >

 Do you want to generate a .module file (yes/no) [yes]:
 > no

 Define module as feature (yes/no) [no]:
 >

 Do you want to add a composer.json file to your module (yes/no) [yes]:
 > no

 Would you like to add module dependencies (yes/no) [no]:
 >


 Do you confirm generation? (yes/no) [yes]:
 >

Generated or updated files
 Site path: /var/www/html
 1 - modules/custom/myroute/myroute.info.yml
```

Let's write a simple controller which prints "hello world" when we hit the path `/hello`.

```
$ drupal generate:controller

 Enter the module name [email_management]:
 > myroute

 Enter the Controller class name [DefaultController]:
 > HelloWorldController

 Enter the Controller method title (to stop adding more methods, leave this empt\
y) [ ]:
 > Hello World

 Enter the action method name [hello]:
```

```
 >

 Enter the route path [/myroute/hello/{name}]:
 > hello

 Enter the Controller method title (to stop adding more methods, leave this empt\
y) [ ]:
 >

 Do you want to generate a unit test class (yes/no) [yes]:
 > no


 Do you want to load services from the container (yes/no) [no]:
 > no


 Do you confirm generation? (yes/no) [yes]:
 >

Generated or updated files
 Site path: /var/www/html
 1 - modules/custom/myroute/src/Controller/HelloWorldController.php
 2 - modules/custom/myroute/myroute.routing.yml
 // router:rebuild

 Rebuilding routes, wait a moment please


 [OK] Done rebuilding route(s).
```

Make sure you enable the module.

```
$ drush en myroute -y
```

Open `modules/custom/myroute/src/Controller/HelloWorldController.php` and change the markup text to “Hello World”.

```
public function hello() {
  return [
    '#type' => 'markup',
    '#markup' => $this->t('Hello World')
  ];
}
```

you might need to rebuild cache.

```
$ drush cr
```

Hit `/hello`.

The equivalent Drupal 7 code for this would be something on the lines of

```
// inside myroute.module...

function myroute_menu() {
  $items = array();
  $items['main'] = array(
    'title' => Hello World',
    'page callback' => myroute_hello',
    'access arguments' => array('access content'),
    'type' => MENU_NORMAL_ITEM,
    'file' => 'myroute.pages.inc'
  );
  return $items;
}

// inside myroute.pages.inc

function myroute_hello() {
  return t(â€˜Hello Worldâ€™);
}
```

Routes with parameters

This is great, but how to add URL parameters?

Let's add a new route with a URL parameter.

```
$ cd myroute
$ git checkout -f route-with-params
```

or if you choose to use drupal console,

```
drupal generate:controller
 Enter the module name [email_management]:
 > myroute

 Enter the Controller class name [DefaultController]:
 > GreetingController

 Enter the Controller method title (to stop adding more methods, leave this empt\
y) [ ]:
 > Greeting

 Enter the action method name [hello]:
 > greeting

 Enter the route path [/myroute/hello/{name}]:
 > hello/{name}

 Enter the Controller method title (to stop adding more methods, leave this empt\
y) [ ]:
 >

 Do you want to generate a unit test class (yes/no) [yes]:
 > no


 Do you want to load services from the container (yes/no) [no]:
 >


 Do you confirm generation? (yes/no) [yes]:
 >

Generated or updated files
 Site path: /var/www/html
 1 - modules/custom/myroute/src/Controller/GreetingController.php
 2 - modules/custom/myroute/myroute.routing.yml
 // router:rebuild
```

```
 Rebuilding routes, wait a moment please


 [OK] Done rebuilding route(s).
```

Edit the greeting controller function to print the name.

```
public function greeting($name) {
  return [
    '#type' => 'markup',
    '#markup' => $this->t('Hello, @name!', ['@name' => $name]),
  ];
}
```

Try `/hello/Joe`.

`/hello/123-456` works too. Let's tweak it a little and add a validation criteria for the parameter, i.e. names can only have alphabets and spaces.

```
myroute.greeting_controller_greeting:
  path: 'hello/{name}'
  defaults:
    _controller: '\Drupal\myroute\Controller\GreetingController::greeting'
    _title: 'Greeting'
  requirements:
    _permission: 'access content'
    name: '[a-zA-z ]+'
```

Rebuild the cache for this to take effect.

```
$ drush cr
```

`/hello/123-456` will now give a 404. Try `/hello/Joe%20Pesci`.

Custom permissions

How about adding custom permissions? What if we want to show the /hello page only to users who can administer content.

```
$ cd myroute
$ git checkout -f custom-access-check
```

We first indicate that the route uses custom permissions.

```
myroute.hello_world_controller_hello:
  path: 'hello'
  defaults:
    _controller: '\Drupal\myroute\Controller\HelloWorldController::hello'
    _title: 'Hello World'
  requirements:
    _custom_access: '\Drupal\myroute\Controller\HelloWorldController::custom_acc\
ess_check'
```

Note that the `_permission` parameter has been replaced by `_custom_access` parameter. Lets implement this custom access method inside the `HelloWorldController`.

```
// add the required namespaces at the top.
use Drupal\Core\Session\AccountInterface;
use Drupal\Core\Access\AccessResult;

  /**
   * Custom access check
   *
   * @param \Drupal\Core\Session\AccountInterface $account
   *   access checking done against this account.
   */
  public function custom_access_check(AccountInterface $account) {
    return AccessResult::allowedIf($account->hasPermission('access content') &&
      $account->hasPermission('administer content'));
  }
```

Rebuild cache and try hitting `/hello` as an anon user, you should get an "Access denied" page. Of note here is the `AccessResult` class, which was introduced[2] to make entity related access checks to be cacheable. We could also do role-based access checks, as in:

```
requirements:
  _role: 'administrator'
```

[2] https://www.drupal.org/node/2337377

Dynamic routes

Defining routes via a YAML file applies to static paths, where we know the routes beforehand. If we want to define a route programmatically, we have to define and return a `\Symfony\Component\Routing\Route` object.

```
$ cd myroute
$ git checkout -f dynamic-routes
```

To illustrate the concept of dynamic routes, let's take an imaginary requirement where we have a custom path, `/content_types/{content_type}`, which will display all the fields of a content type `{content_type}`. In order to create a dynamic route for the same, we have to create our own `Routing` class inside the `src/Routing` directory and override the `routes()` method. This is equivalent to its YAML cousin, the `myroute.routing.yml` file, but written in PHP.

```
<?php

/**
 * @file
 * Contains \Drupal\myroute\Routing\CTRoutes.
 */

namespace Drupal\myroute\Routing;

use Symfony\Component\Routing\Route;

/**
 * Dynamic routes for content types.
 */
class CTRoutes {

  /**
   * {@inheritdoc}
   */
  public function routes() {
    $routes = [];
    $content_types = \Drupal::service('entity.manager')->getStorage('node_type')\
->loadMultiple();
    foreach ($content_types as $content_type) {
      $routes['myroute.content_type_controller.' . $content_type->id() ] = new R\
oute(
```

```
        '/content_types/' . $content_type->id(),
        array(
          '_controller' => '\Drupal\myroute\Controller\CTController::fields',
          '_title' => 'Field info for ' . $content_type->label(),
          'content_type' => $content_type->id(),
        ),
        array(
          '_permission'  => 'access content',
        )
      );
    }
    return $routes;
  }
}
```

The custom router above creates an array of routes for every content type in the system and returns it. For instance, if there are 3 content types in the system, like page, foo and bar, there will be 3 routes with paths `/content_types/page`, `/content_types/foo` and `/content_types/bar` respectively.

Let's flesh out the `CTController` as well for this dynamic route.

```
<?php

namespace Drupal\myroute\Controller;

use Drupal\Core\Controller\ControllerBase;

/**
 * Class CTController.
 *
 * @package Drupal\myroute\Controller
 */
class CTController extends ControllerBase {

  /**
   * List fields info of a content type.
   *
   * @return string
   *   Return field list.
   */
  public function fields($content_type) {
    $render = '<table><tr><th>' . $this->t('Field type') . '</th><th>' . $this->\
t('Label') . '</th></tr>';
```

```
    $field_definitions = \Drupal::entityManager()->getFieldDefinitions('node', $\
content_type);
    foreach ($field_definitions as $field_name => $field_definition) {
      if (!empty($field_definition->getTargetBundle())) {
        $render .= '<tr><td>' . $field_definition->getType() . '</td><td>' . $fi\
eld_definition->getLabel() . '</td></tr>';
      }
    }
    $render .= '</table>';
    return [
      '#type' => 'markup',
      '#markup' => $render,
    ];
  }
}
```

The `field()` method looks up the field definitions of the `content_type` argument and renders it as a HTML table.

Finally, we have to indicate Drupal to pick up the custom Routing class we've added. This can be done by adding a `_route_callbacks` item to the routing YAML file.

```
route_callbacks:
  - '\Drupal\myroute\Routing\CTRoutes::routes'
```

Rebuild the cache and you are all set. Hit the `content_types/article` page, you should see something like this(provided you have the article CT in your system).

Article CT listing

Controllers

A controller is a PHP callable(function, or method of an object), but its usually a method inside a controller class in both Symfony and Drupal 8. While Symfony returns a `Response` object in a controller, Drupal 8 returns a render array. What exactly constitutes a controller class? It contains the glue code between a route and the business logic. Many a times, we end up writing a lot of boilerplate code inside our controllers. For instance, we might need information about the current user in order to modify the response accordingly. Another example would be using the configuration API inside the controller to get a configuration object. This is such a common pattern, that Drupal provides

us with an abstract class called `ControllerBase`. This class provides a lot of useful methods which could be used in controllers. For example, the above 2 requirements are defined in `ControllerBase`.

A method to get the current user.

```
protected function currentUser() {
  if (!$this->currentUser) {
    $this->currentUser = $this->container()->get('current_user');
  }
  return $this->currentUser;
}
```

Another method to get config object.

```
protected function config($name) {
  if (!$this->configFactory) {
    $this->configFactory = $this->container()->get('config.factory');
  }
  return $this->configFactory->get($name);
}
```

That said, you don't need everything `ControllerBase` brings if you are not using it. It could be a trivial method inside a class. Let's say you want to return a list of all the public Github repositories of a user as JSON, for the route `/repos/{handle}`, it could look like this:

A common anti-pattern is to put all the business logic related code inside controllers. Let's take an example scenario where we have to show a list of all the Github repositories of the currently logged in user in the route `/repos`.

2. Forms

[TODO]