Learn Drupal 8

02/10/2018

Plan

Plan

Plan

When to use a contrib module

- Generic functionality not specifically bound to client
- Mostly hosted on packages.drupal.org
- Use composer to resolve dependecies
- Famous contrib modules now included in core since v8.x:
  - Views
  - Entity Reference
  - Media + Entity Browser
  - Content Moderation + Workflows

Plan

Practice

- @see from practice document:
- Go to Drupal.org, find the modules search page
- Identify the install vs enabled ratio on any module page

Plan

Well known contrib modules

- Commonly used community modules for administrators:
  - Admin Toolbar: optimized JS administration menu
  - Field Group: adds fieldsets and container to entity fields
  - Paragraphs: adds dynamic field
- Modules for SEO enhancements:
  - Google Analytics
  - Metatag + Token: override meta such as title and description
  - Redirect + Pathauto: manage url alias and 301 redirections
- Modules for search engines:
  - Search Api, Solr, Elasticsearch Connector: enhanced fulltext search solutions
  - Facets: multi criteria facetted search menu

Plan

Practice

- @todo from practice document:
  - Install admin toolbar, log in and view log reports
  - Add an article field as an entity relationship to basic page
  - Install the Paragraphs module, create a paragraph and bind it to basic page
  - Add a collapsible field group fieldset containing the above fields

Plan

Modules

When to create a custom module?

- No community module meets your needs
  - Don't take a module that covers only 20% of your needs
  - Many modules always have a cost on performances
- Specific needs connected to the client business:
  - Dynamic block (Plugin)
  - Specific page (Symfony Route)
  - Customize entities (Entity API)
- Alter the behavior of an existing module or core
  - Change a form for example
  - Add an action when an event appears
    - Send a mail when publishing a node for example

Plan

Modules - Location

Unlike Drupal 7, in Drupal 8, modules and themes must be placed in the Drupal root folders

Modules - Location

Unlike Drupal 7, in Drupal 8, modules and themes must be placed in the Drupal root folders

- `/modules`
  - Usage of subdirectories "contrib", "custom" and "patches" is best practice
- `/themes`
  - Usage of a subdirectory "custom" is best practice
- `/profiles`

Modules - Location
Multisite

In case of multisites

- `/modules` are available for all sites
  - `sites/your_site_name/modules` allow to restrict modules to one site
- `/themes` are available for all sites
  - `sites/your_site_name/themes` possible
- `/profiles` are available for all sites
  - `sites/your_site_name/profiles` possible

Plan

Modules - Minimal structure

```
|—— my_module.info.yml
|—— my_module.module
```

- The `module_name.info.yml` is the only file needed to declare and activate a module
- info.yml properties list:
  - title (required)
  - type (required): this is either a module or theme
  - description (optionnal)
  - core (required): compatible core version, often 8.x
  - version (optionnal): this module version, generally dynamic VERSION
  - dependencies (optionnal): modules needed to install this module

Plan

Practice

- @todo from practice document:
  - Learn to use Devel
  - Download and Patch the Broken Link a module
  - Create an empty module

Modules - Common structure

```
|—— config
    |—— install
    |—— schema
    |—— ...
|—— css
|—— js
|—— src
    |—— Controller
    |—— Plugin
    |—— Form
    |—— Entity
    |—— ...
|—— templates
|—— tests
|—— my_module.info.yml
|—— my_module.module
|—— my_module.install
|—— my_module.libraries.yml
|—— my_module.actions.yml
|—— my_module.routing.yml
|—— my_module.services.yml
```

- `config` contains the stored configuration (in yml) of your module
- `src` folder contains classes
- `module_name.install` is used to do some operation on install/uninstall
- `*.libraries.yml` contains libraries like js or css
- `*.routing.yml` contains the routes
- `*.services.yml` contains all the module services and override

Plan

Plan

Plugins - Definition

- Plugins are a general reusable solution to a recurring problem within a given context
- Implement different behaviors via a common interface
- Plugins that perform similar functionality are of the same plugin type

Plugins - Definition

- Plugins are a general reusable solution to a recurring problem within a given context
- Implement different behaviors via a common interface
- Plugins that perform similar functionality are of the same plugin type
- Commonly used example of plugins
  - Create blocks
  - Field formatters and field form widgets
  - Image stlye effect (scale, rotate...)
  - Views handlers (fields, filters, contextual arguments)
  - QueueWorker (waiting lists triggered by cron)
  - CKEditor plugins

Plugins - Definition

- Plugins are a general reusable solution to a recurring problem within a given context
- Implement different behaviors via a common interface
- Plugins that perform similar functionality are of the same plugin type
- Commonly used example of plugins
  - Create blocks
  - Field formatters and field form widgets
  - Image stlye effect (scale, rotate...)
  - Views handlers (fields, filters, contextual arguments)
  - QueueWorker (waiting lists triggered by cron)
  - CKEditor plugins
- And more !
- /! Tip: reading contrib code source is often helpful.

Plan

Plugins - Plugin type

- Plugin types are contract classes for a generic need.
  - Implement interfaces: build your class with required methods
  - Extend classes: reuse specific functionality

Plugins - Plugin type

- Plugin types are contract classes for a generic need.
  - Implement interfaces: build your class with required methods
  - Extend classes: reuse specific functionality
- It ensures that the plugin manager will know how to interface with your plugin
  - to get the data it needs
  - to ask it to perform a given operation (ex: a build() method for rendering blocks)

Plugins - Plugin type

- Plugin types are contract classes for a generic need.
  - Implement interfaces: build your class with required methods
  - Extend classes: reuse specific functionality
- It ensures that the plugin manager will know how to interface with your plugin
  - to get the data it needs
  - to ask it to perform a given operation (ex: a build() method for rendering blocks)
- Some commonly used plugin types
  - `Block`
    - Create custom blocks
  - `FieldFormatter`, `FieldWidget`
    - Providing custom field API display format and field form widgets
  - `Views Plugins`
    - ViewsField, ViewsFilter, ViewsArgument

Plan

Plugins - Plugin discovery

Plugin discovery

- Annotation
  - Plugin classes are annotated and placed in a defined namespace subdirectory (`*/src/Plugin/*/*`)
  - Most Drupal Core plugins use this method of discovery and it's the recommended method
- YAML
  - Some plugins are listed in YAML files
  - Drupal Core uses this method for discovering local tasks and local actions

Plan

Plugins - Create a new Plugin

- Plugin are created for a specific `Plugin type` (see later)
- Create a class in `*/src/Plugin/[Plugin type]/`
  - The class name generally ends with `[Plugin type]`
- Implements the plugin type interface
  - If provided prefer extends your class with a `[Plugin type]Base` class
- Set the correct annotation in the class header

```
* Provides a [Plugin type] plugin.
*
* @[Plugin type](
*  id = "myplugin_id",
*  admin_label = @Translation("My Class Plugin"),
* )
*/
class MyClass[Plugin type] extends [Plugin type]Base {
//...
}
```

Plugins - Create a new Plugin

- Example with a Block plugin type

```
/**
 * Provides a 'TrainingBlock' block.
 *
 * @Block(
 *   id = "training_block",
 *   admin_label = @Translation("Training block"),
 * )
 */
class TrainingBlock extends BlockBase {
//...
}
```

Note

- It can be generated using Drupal Console
  - `generate:plugin:[plugin type]`
  - `generate:plugin:block`

Plan

Plugins - YAML Plugins

- Create links with `*.links.(type).yml` files
  - `module_name.links.task.yml` (backoffice links tabs)
  - `module_name.links.menu.yml` (front links)
  - `module_name.links.action.yml` (backoffice links buttons)
  - `module_name.links.task.yml` (backoffice links tabs)
- Create new User permissions with `*.permissions.yml` files
  - `module_name.permissions.yml` (front links)
    https://api.drupal.org/api/drupal/core

Plugins - YAML Plugins
module_name.links.menu.yml

`module_name.links.menu.yml` declares

- menu metadatas (title, description, ...)
- target (route name)
- weight
- hierarchy

Plugins - YAML Plugins

module_name.links.menu.yml

- **title (required)**: Untranslated title
- **description (optional)**: Untranslated description
- **route_name/route_parameters (optional)**: Route name/parameters to be used to build the path
  - Either the `route_name` or `url` element must be provided
- **url (optional)**: An external URL
- **parent (optional)**: The parent menu id
- **weight (optional)**: Position of the items in menu (Default 0)
- **expanded (optional)**: TRUE to expand the menu

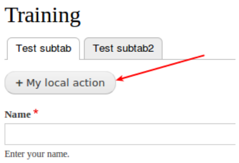
Plugins - YAML Plugins

module_name.links.menu.yml

```
# Define some menu links for an entity related module
entity.content_entity_example_contact.collection:
  title: 'Content Entity Example: Contacts Listing'
  route_name: entity.content_entity_example_contact.collection
  description: 'List Contacts'
  weight: 10

content_entity_example_contact.admin.structure.settings:
  title: Contact Settings
  description: 'Configure Contact entity'
  route_name:  content_entity_example.contact_settings
  parent: system.admin_structure
```

Plugins - YAML Plugins
module_name.links.action.yml


Training

Name *

Enter your name.

Local actions are items that describe actions on the parent item such as adding a new user or block

- Similar to Drupal 7 "`MENU_LOCAL_ACTION`"

Plugins - YAML Plugins

module_name.links.action.yml

```yaml
# All action links for this module
training.action:
  title: 'My local action'
  route_name: training.myroute
  # Where will the link appear, defined by route name.
  appears_on:
    - training.route
```

Practice

- @todo from practice document
  - add an action link plugin in your module from given instructions
  - add a permission plugin which will be used for coming exercises

Plugins - YAML Plugins

Creating a new Plugin type

- As creating a new plugin type is not a common task please visit the page for more detail: `https://api.drupal.org/api/drupal/core%21core.api.php/group/plugin_api/8#sec_define`

Note

- It can be generated using Drupal Console
  - `generate:plugin:type:annotation`

Plugins
More info on Plugin API

More info about Plugin API

https://api.drupal.org/api/drupal/core%21core.api.php/group/plugin_api/8

https://www.sitepoint.com/drupal-8-custom-plugin-types/

Plan

Plan

Blocks - Definition

- Blocks are elements to render HTML

Blocks - Definition

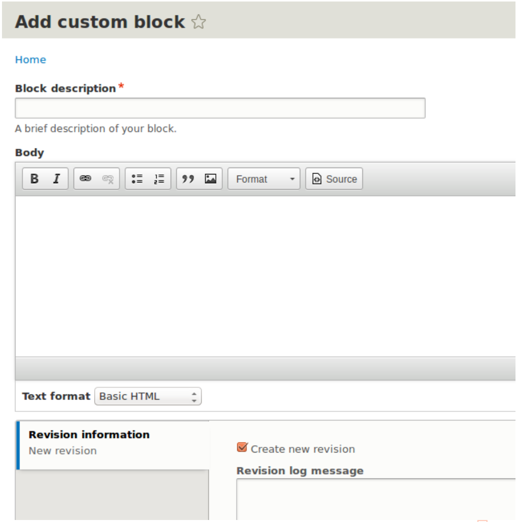
- Blocks are elements to render HTML
- A block can be assigned to one or several region

Blocks - Definition

- Blocks are elements to render HTML
- A block can be assigned to one or several region
- Internally
  - A configuration entity store the instance placement informations: theme, region, weight
  - Plugins are used to manage the block rendering and contextualisation (conditions)
    - Folder `*/src/Plugin/Block/*`
    - Folder `*/src/Plugin/Condition/*`

Blocks - Definition
From backoffice

- From the backoffice /block/add
- You can create custom blocks (fieldable entities)



Add custom block ☆

Home

Block description *

A brief description of your block.

Body

Text format Basic HTML

Revision information
New revision

✓ Create new revision

Revision log message

Blocks - Definition

With code

- To create block by code, you must:
  - Create a new class in `*/src/Plugin/Block/*`
    - that implements the `\Drupal\Core\Block\BlockPluginInterface`
    - or that inherit from `\Drupal\Core\Block\BlockBase`
  - Register your plugin with an annotation in your class header
    - Annotation declared by `\Drupal\Core\Block\Annotation\Block`

Blocks - Definition
With code

Code example

```
namespace Drupal\yourmodule\Plugin\Block;

use Drupal\Core\Block\BlockBase;

/**
 * Provides my custom block.
 *
 * @Block(
 *   id = "my_custom_block",
 *   admin_label = @Translation("My Custom Block"),
 *   category = @Translation("Blocks")
 * )
 */
class YourBlockNameBlock extends BlockBase {

  /**
   * {@inheritdoc}
   */
  public function build() {
    $render = [
      '#markup' => 'hello world'
    ];
    return;
  }

}
```

Plan

Blocks - Rendering

- In Drupal 8 like in Drupal 7, Drupal use Render arrays to render elements
- It gives you complete control over the site appearence
- Hierarchical associative array containing data to be rendered
  - It is mandatory, you cannot return HTML directly
  - Can render Twig templates or HTML elements

Blocks - Rendering
Render arrays

- A render array must have one of the following main properties
  - `#type` form, textfield, submit...
  - `#theme` theme to render (declared with hook_theme())
  - `#markup` direct HTML
- `#prefix/#suffix` to add some element before and/or after the current element
- `#attached` property is to attach library (css/js)
- `#cache`
- `https://www.drupal.org/developing/api/8/render/arrays`

Plan

Blocks - Practice

Practice1

- @todo form practice document
  - Create a block plugin displaying the following message:
    - current user name
    - current route name
    - number of url parameters
  - Play with block configuration in the backoffice:
    - place your block above page title
    - duplicate it in another region
    - use contexts: display only on homepage (<front>)
  - Verify the cache is working properly (activate your cache settings)

Plan

Plan

What is a route?

A route is a path defined for Drupal to route users toward some page content.
There is a route for every page of the website.

What is a route?

A route is a path defined for Drupal to route users toward some page content.
There is a route for every page of the website. Some example of routes:

- `/node`
- `/user/login`
- `/admin/config`

What is a route?

A route is a path defined for Drupal to route users toward some page content.
There is a route for every page of the website. Some example of routes:

- `/node`
- `/user/login`
- `/admin/config`

Flow for a request:

- Drupal check if the route exists
  - If route exists, Drupal use the route's definitions to display content
  - If not, Drupal serve the 404 route.

Routing - Routing

web/core/modules/system/system.routing.yml

```
system.404:
  path: '/system/404'
  defaults:
    _controller: '\Drupal\system\Controller\Http4xxController:on404'
    _title: 'Page not found'
  requirements:
    _access: 'TRUE'
```

1. route "`system.404`" is mapped to the URI "`/system/404`"
   - Best practice: prefix with module name
2. accessing path "`/system/404`" will check the "`access content`" permission
3. `Http4xxController::on404()` method is invoked from controller

Class autoloading (namespace resolution)

- Core class autoloading (block, comment, field, node, taxonomy, views etc ...): `\Drupal\system\Controller\Http4xxController:on404`
- will search in `core/modules/system/src/Controller/Http4xxController.php`
- Custom and Contrib Class autoloading: `\Drupal\example\Controller\CustomController:method`
- will search in `modules/*/example/src/Controller/CustomController.php`

Required route properties

- **path**:
  - URL to the route, with a leading forward slash
    ie: `path: '/book'`
  - Possible arguments with curly braces.
    ie: `path: '/node/{node}/test'`
- **defaults**:
  - default properties of a route (`_controller` or `_form`, and `_title`)
- **requirements (required)**
  - conditions to verify to make access possible

Defaults route properties

- **_controller**: value is a callable (ie: classname::method)
  - return a renderable array or a `Symfony\Component\HttpFoundation\Response` object
- **_form**: will be used to render a form directly
  - a class implementing `Drupal\Core\Form\FormInterface` is expected
- **_title** (optional): Page title for the route

Routing - Routing
Requirements

Requirements
Determines what conditions must be verified in order to grant access to the route.

- **_permission**: a permission string
    - multiple permissions by separating them with ',' for AND logic
    - '+' for OR logic
    - examples
        - `_permission: 'access content'`
        - `_permission: 'access content, administer node'`
- **_role**: access possible for some roles
    - same logic for ',' and '+'

Routing - Routing

Many configurations are possible for routing,
for more informations visit

`https://www.drupal.org/node/2092643`

and

`https://symfony.com/doc/3.4/routing.html`

Practice

- @todo from practice document:
  - Prepare a route with ImagesWallController class returning a render array.
  - Limit access to the permission you created previously
  - Hint: Copy the code from the previous 404 controller example from core and make it work in your module.

Plan

Plan

What is a service?

- A `service` is a PHP Object that performs some "global" task.
- Instanciated once by Service Container (Singleton Pattern)

What is a service?

- A `service` is a PHP Object that performs some "global" task.
- Instanciated once by Service Container (Singleton Pattern)
- Example of service
  - Database access
  - Sending Email
  - Logs
  - Routing
  - Translation
  - Theme
  - ...

Plan

Services - Practice
Drupal embed services

- List of important services directly callable from the Drupal class:
  - cache
  - config
  - currentUser
  - database
  - httpClient
  - logger
  - request
  - routeMatch
- @see `https://api.drupal.org/api/drupal/core!lib!Drupal.php/class/Drupal/8.6.x`

Plan

Services - Debug existing services

- You can find the available services using the following command line

```
drupal debug:container

 Service ID                        Class Name
 class_loader                      Symfony\Component\ClassLoader\ApcClassLoader
 kernel                            Drupal\Core\DrupalKernel
 service_container                 Drupal\Core\DependencyInjection\Container
 cache_context.ip                  Drupal\Core\Cache\Context\IpCacheContext
 cache_context.headers             Drupal\Core\Cache\Context\HeadersCacheContext
 cache_context.cookies             Drupal\Core\Cache\Context\CookiesCacheContext
 cache_context.session             Drupal\Core\Cache\Context\SessionCacheContext
 cache_context.session.exists      Drupal\Core\Cache\Context\SessionExistsCacheContext
 ...
```

Services - Debug existing services

- You can declare a custom service in `my_module.services.yml`
- You can easily override any services keeping the same service name
  - https://www.drupal.org/docs/8/api/services-and-dependency-injection/altering-existing-services-providing-dynamic-services

Services - Arguments

my_module.services.yml

```
services:
  user.current_user_context:
    class: Drupal\user\ContextProvider\CurrentUserContext
    arguments: ['@current_user', '@entity.manager']
```

- `Drupal\user\ContextProvider\CurrentUserContext` will be instanciated
- Injection of the services' instance
  - `current_user`
  - `entity.manager`

Note

The "@" allows to pass a service instance to another service.

Services - Arguments

Optional arguments

You can push optional arguments

- use the "@?" syntax to pass a service only if it exists

```
services:
    newsletter_manager:
        class:     Acme\HelloBundle\Newsletter\NewsletterManager
        arguments: ["@?my_mailer"]
```

Services - Arguments
Optional arguments

You can push optional arguments

- use the "@?" syntax to pass a service only if it exists

```
services:
    newsletter_manager:
        class:     Acme\HelloBundle\Newsletter\NewsletterManager
        arguments: ["@?my_mailer"]
```

- The NewsletterManager class will get the `my_mailer` only if it exists

```
public function __construct(Mailer $mailer = null)
{
    //...
}
```

Plan

Services - Arguments

Injecting a service in a controller

- Controllers implements the ContainerInjectionInterface for dependency injection.

```
use Drupal\Core\DependencyInjection\ContainerInjectionInterface;

class RouteController extends ControllerBase implements ContainerInjectionInterface {

  private $pixabayRequester;

  /**
   * {@inheritdoc}
   */
  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('d8_lille.pixabay_requester')
    );
  }

  public function __construct(PixabayRequester $pixabayRequester) {
    $this->pixabayRequester = $pixabayRequester;
  }
```

Services - Usage

As we saw previously, services can be injected but it's also possible to get one anywhere with the helper:

```
$pixabayRequester = \Drupal::service('d8_lille.pixabay_requester');
```

Services - Arguments

Injecting a service in a controller

- Or either:

```
use Drupal\Core\DependencyInjection\ContainerInjectionInterface;

class RouteController extends ControllerBase {

  private $pixabayRequester;

  public function __construct(PixabayRequester $pixabayRequester) {
    $this->pixabayRequester = \Drupal::service('d8_lille.pixabay_requester');
  }
```

Practice

- @todo from practice document:
  - Prepare the PixabayRequester service class above and inject it in your controller
  - Inject the Guzzle Drupal service into PixabayRequester to retrieve images from pixabay