Drupal 8 Intro

Peter M. Wolanin

Momentum Specialist (principal engineer), Acquia, Inc.

Drupal contributor drupal.org/user/49851

February 02, 2014

Here’s What’s In the Talk

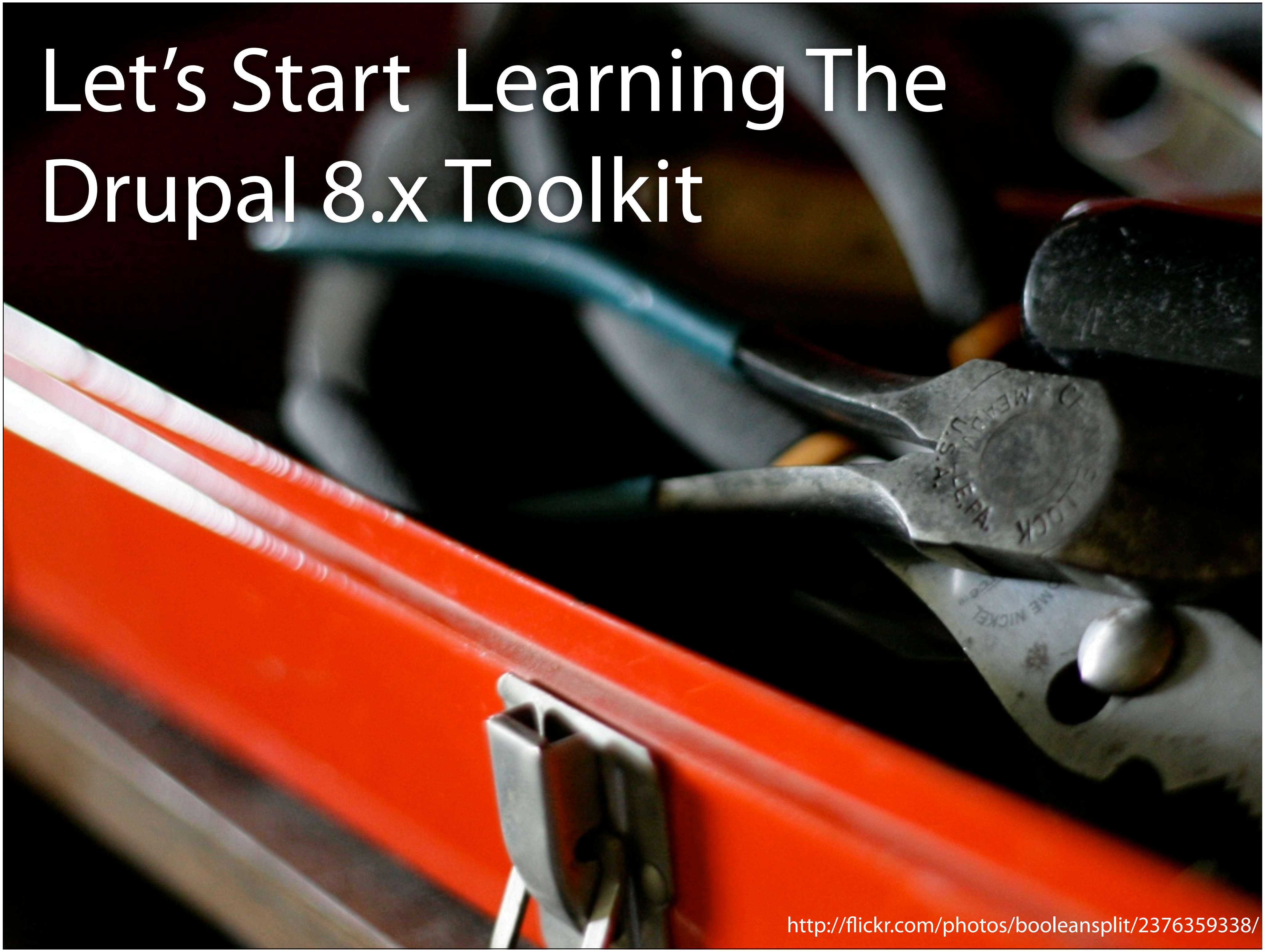
- Background
- Basics of a module in Drupal 8
- Simple but common plugin example
  - Adding new tabs (a.ka. local tasks)
- “What’s a plugin?”
- Example of a core info hook conversion
- Configurable plugins (ConfigEntity based)
  - Adding a custom block

Drupal 8 Background

- I’ll assume you know something about:
  - The DIC/container/service container - an object that contains instances of “services” (the current request, the current user, URL generator, etc).
- The new routing system - names instead of paths.
  - a route name is just a machine name that connects to a path pattern, callbacks to provide, title, content, access etc. - like a D7 menu router.
- Namespaced classes (PHP 5.3+) like `\Drupal\search\Plugin\Block\SearchBlock`

Learn More About the DIC

- https://portland2013.drupal.org/session/dependency-injection-drupal-8
- Look at all the services.yml files in Drupal 8
- http://symfony.com/doc/master/components/dependency_injection/index.html



Let’s Start Learning The Drupal 8.x Toolkit

http://flickr.com/photos/booleansplit/2376359338/

A Drupal 8 Module

- As in Drupal 7, blocks are provided by modules - so you need a module. You need a .info.yml file and an (empty) .module file.

modules/mymodule/mymodule.info.yml

```
name: 'My test module'
type: module
description: 'Drupalcon demo.'
core: 8.x
```

modules/mymodule/mymodule.module

```
<?php

/**
 * @file
 * Drupalcon demo module.
 */
```

Add Routes:

- routes need to be defined by your module: **mymodule/mymodule.routing.yml**

```
mymodule.list:
  path: '/admin/config/mymodule/list'
  defaults:
    _content: '\Drupal\mymodule\Controller\MyController::dolist'
    _title: 'Mymodule list'
  requirements:
    _access: 'TRUE'

mymodule.settings:
  path: '/admin/config/also-mymodule/settings'
  defaults:
    _content: '\Drupal\mymodule\Controller\MyController::settings'
    _title: 'Mymodule settings'
  requirements:
    _access: 'TRUE'
```

Adding Two Tabs:

- For most uses, just add a YAML file listing your tabs: **mymodule/mymodule.local_tasks.yml**

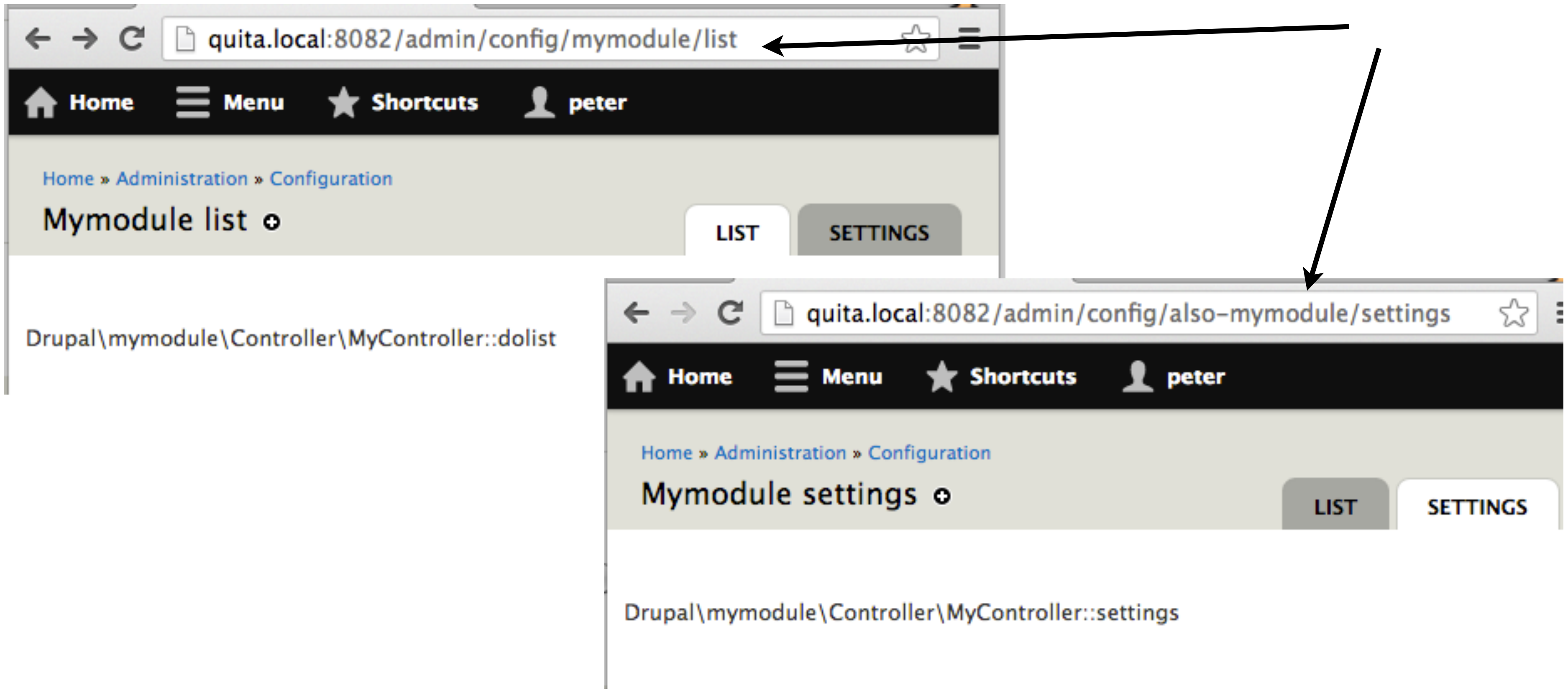
```
mymodule.list_tab:
  route_name: mymodule.list
  title: 'List'
  base_route: mymodule.list

mymodule.settings_tab:
  route_name: mymodule.settings
  title: 'Settings'
  base_route: mymodule.list
```

Plugin ID

Local tasks reference one route as the "base" that anchors them

Adding Two Tabs:

- Unlike Drupal 7 you don't need to jump though the hoops of a default local task, or making the paths align in a certain hierarchy


quita.local:8082/admin/config/mymodule/list
Home
Menu
Shortcuts
peter
Home » Administration » Configuration
Mymodule list
LIST
SETTINGS

Drupal\mymodule\Controller\MyController::dolist

quita.local:8082/admin/config/also-mymodule/settings
Home
Menu
Shortcuts
peter
Home » Administration » Configuration
Mymodule settings
LIST
SETTINGS

Drupal\mymodule\Controller\MyController::settings

LocalTask Plugin Keys:

The plugin configuration options and defaults are on the LocalTaskManager class

```
class LocalTaskManager extends DefaultPluginManager {
  protected $defaults = array(
    // (required) The name of the route this task links to.
    'route_name' => '',
    // Parameters for route variables when generating a link.
    'route_parameters' => array(),
    // The static title for the local task.
    'title' => '',
    // The route name where the root tab appears.
    'base_route' => '',
    // The plugin ID of the parent tab (or NULL for the top-level tab).
    'parent_id' => NULL,
    // The weight of the tab.
    'weight' => NULL,
    // The default link options.
    'options' => array(),
    // Default class for local task implementations.
    'class' => 'Drupal\Core\Menu\LocalTaskDefault',
    // The plugin id. Set by the plugin system based on the top-level YAML key.
    'id' => '',
  );
```

Plugins:

- Encapsulate some re-useable functionality inside a class that implements one or more specific interfaces.
- Plugins combine what in Drupal 7 was an info hook and a number of implementation hooks and possibly configuration: e.g. `hook_search_info()` and `hook_search_execute()`, etc., or `hook_block_info()` and `hook_block_view()`, `_configure()`, `_save()`
- Evolved from ctools and views plugins, but use quite different mechanisms to discover them.

Plugin Manager and IDs

- Every plugin type has a manager - registered as a service (available from the DIC) and used to find and instantiate the desired plugin instance(s).
- Each plugin has an ID, which may be in its definition, or generated as a derivative.
- For a given plugin ID one single class will be used for any plugin instances using that plugin ID.
- A plugin instance is specified by the combination of plugin ID and its configuration values, potentially coming from a ConfigEntity.

7.x: hook_image_toolkits()

```
/**
 * Implements hook_image_toolkits().
 */
function system_image_toolkits() {
  include_once DRUPAL_ROOT . '/' . drupal_get_path('module',
'system') . '/' . 'image.gd.inc';
  return array(
    'gd' => array(
      'title' => t('GD2 image manipulation toolkit'),
      'available' => function_exists('image_gd_check_settings') &&
        image_gd_check_settings(),
    ),
  );
}
```

8.x: ImageToolkitManager

```
class ImageToolkitManager extends DefaultPluginManager {
  // ... various methods ... //

  /**
   * Gets a list of available toolkits.
   */
  public function getAvailableToolkits() {
    // Use plugin system to get list of available toolkits.
    $toolkits = $this->getDefinitions();

    $output = array();
    foreach ($toolkits as $id => $definition) {
      if (call_user_func($definition['class'] . '::isAvailable')) {
        $output[$id] = $definition;
      }
    }
    return $output;
  }
}
```

7.x: desaturate function

```
/**
 * Converts an image to grayscale.
 *
 * @param $image
 *   An image object returned by image_load().
 *
 * @return
 *   TRUE on success, FALSE on failure.
 *
 * @see image_load()
 * @see image_gd_desaturate()
 */
function image_desaturate(stdClass $image) {
  return image_toolkit_invoke('desaturate', $image);
}
```

8.x: desaturate method

```
/**
 * Defines the GD2 toolkit for image manipulation within Drupal.
 *
 * @ImageToolkit(
 *   id = "gd",
 *   title = @Translation("GD2 image manipulation toolkit")
 * )
 */
class GDToolkit extends PluginBase implements ImageToolkitInterface {
  // ... all the toolkit methods ... //

  public function desaturate(ImageInterface $image) {
    // PHP using non-bundled GD does not have imagefilter.
    if (!function_exists('imagefilter')) {
      return FALSE;
    }

    return imagefilter($image->getResource(), IMG_FILTER_GRAYSCALE);
  }
}
```

What about hooks?

http://flickr.com/photos/avelino_maestas/2532426169/

Hooks still have their place:

- Many plugin managers invoke an `_alter` hook so the modules can add to or alter the plugins' definitions. E.g. `hook_block_alter()` allows you to alter the block plugin definitions.
- Info hooks that simply return a data array - like `hook_permission()` - without associated functionality - are not candidates to become plugins.

Plugin Discovery

- The discovery of plugins is basically the same as invoking an info hook (in fact you can implement it that way).
- Discovery gives you an array of plugin definitions, each of which is just an array of keys and values.
- The discovery process fills in defaults, such a `'provider'` which is the name of the module providing the plugin.

Plugin Discovery/Config in Core

- YAML based:

  LocalTask, LocalAction, ContextualLink

- Annotation, some config, but no config entity:

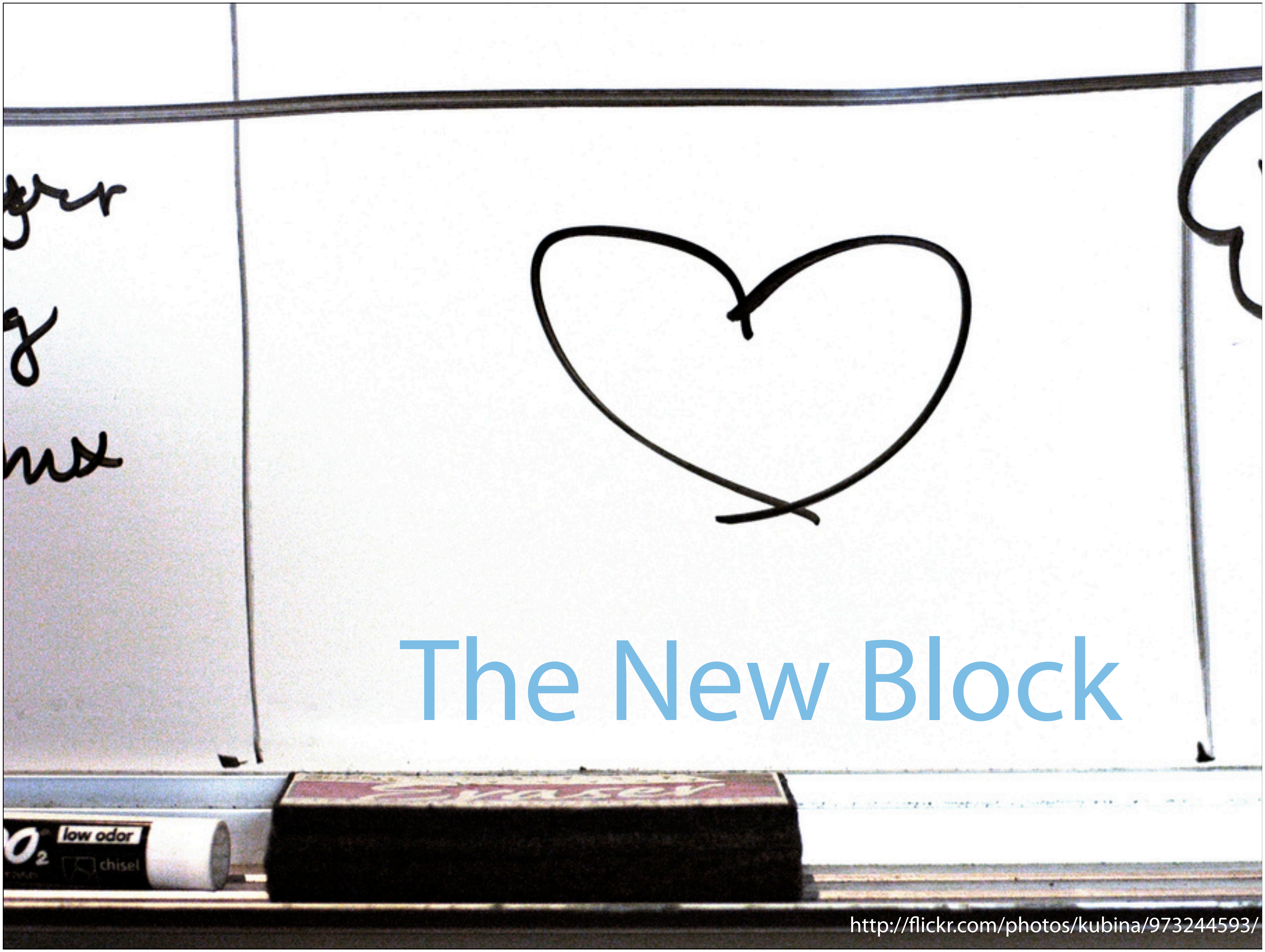
  ImageToolkit, Archiver, StreamWrapper

- Annotation and config entity (many) including:

  Block, ViewsDisplay, SearchPlugin, ImageEffect, Tip, ...

- Not truly a Plugin but uses Annotation discovery:

  Entity (Node, User, etc.)



The New Block

http://flickr.com/photos/kubina/973244593/

Blocks as Plugins

- Each custom block is defined in code as a class.
- When the admin places the block into a region in a theme a configuration object is created to track that setting.
- The config object is a ConfigEntity - it's an abstraction on top of CMI (storing your Drupal configuration in YAML files) - it makes it convenient to list, load, etc. using entity functions. So Drupal can easily list the active block instances.
- Note - you don't need to worry about the config!

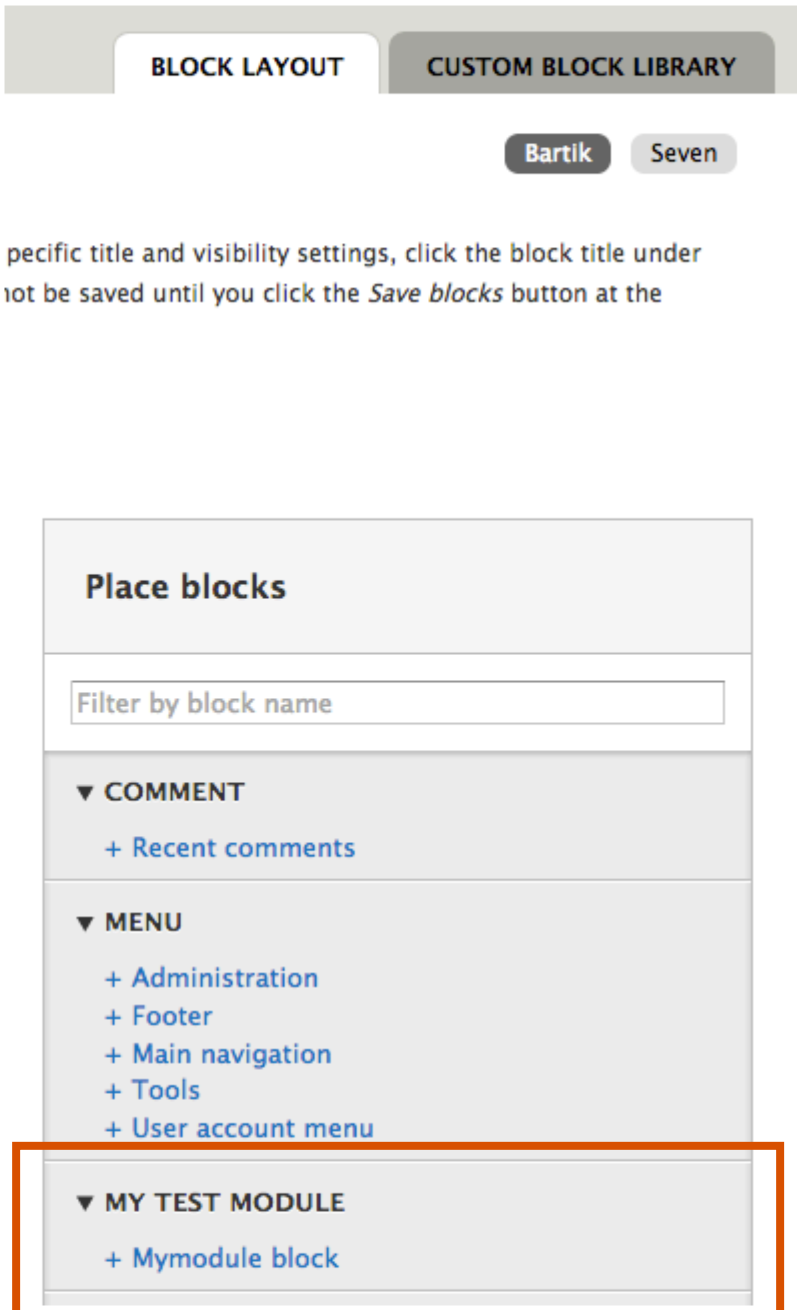
Blocks Implementation

- Blocks implement the `\Drupal\block\BlockPluginInterface`
- If you extend the abstract `\Drupal\block\BlockBase` class then all you need to implement is the `build()` method.
- `build()` is basically the same as `hook_block_view()` in Drupal 7
- For example, I added to my module `\Drupal\mymodule\Plugin\Block\MyBlock`

Side Note - PSR-0/4

- When I add the Block to my module: `\Drupal\mymodule\Plugin\Block\MyBlock`
- This is at the corresponding filepath (under the Drupal root dir): `modules/mymodule/lib/Drupal/mymodule/Plugin/Block/MyBlock.php`
- Yeah, that's long. Note that the full class name and path match under `lib/`
- PSR-4 will be adopted soon and then it will be: `modules/mymodule/lib/Plugin/Block/MyBlock.php`

```
/**
 * Provides a block with 'Mymodule' links.
 *
 * @Block(
 *   id = "mymodule_my_block",
 *   admin_label = @Translation("Mymodule block")
 * )
 */
class MyBlock extends BlockBase {
  public function build() {
    return array(
      'first_link' => array(
        '#type' => 'link',
        '#title' => $this->t('Mymodule List'),
        '#route_name' => 'mymodule.list',
      ),
      'second_link' => array(
        '#type' => 'link',
        '#title' => $this->t('Mymodule Settings'),
        '#route_name' => 'mymodule.settings',
      ));
  }
}
```



BLOCK LAYOUT
CUSTOM BLOCK LIBRARY

Bartik Seven

pecific title and visibility settings, click the block title under
ıot be saved until you click the *Save blocks* button at the

Place blocks

Filter by block name

▼ COMMENT

+ Recent comments

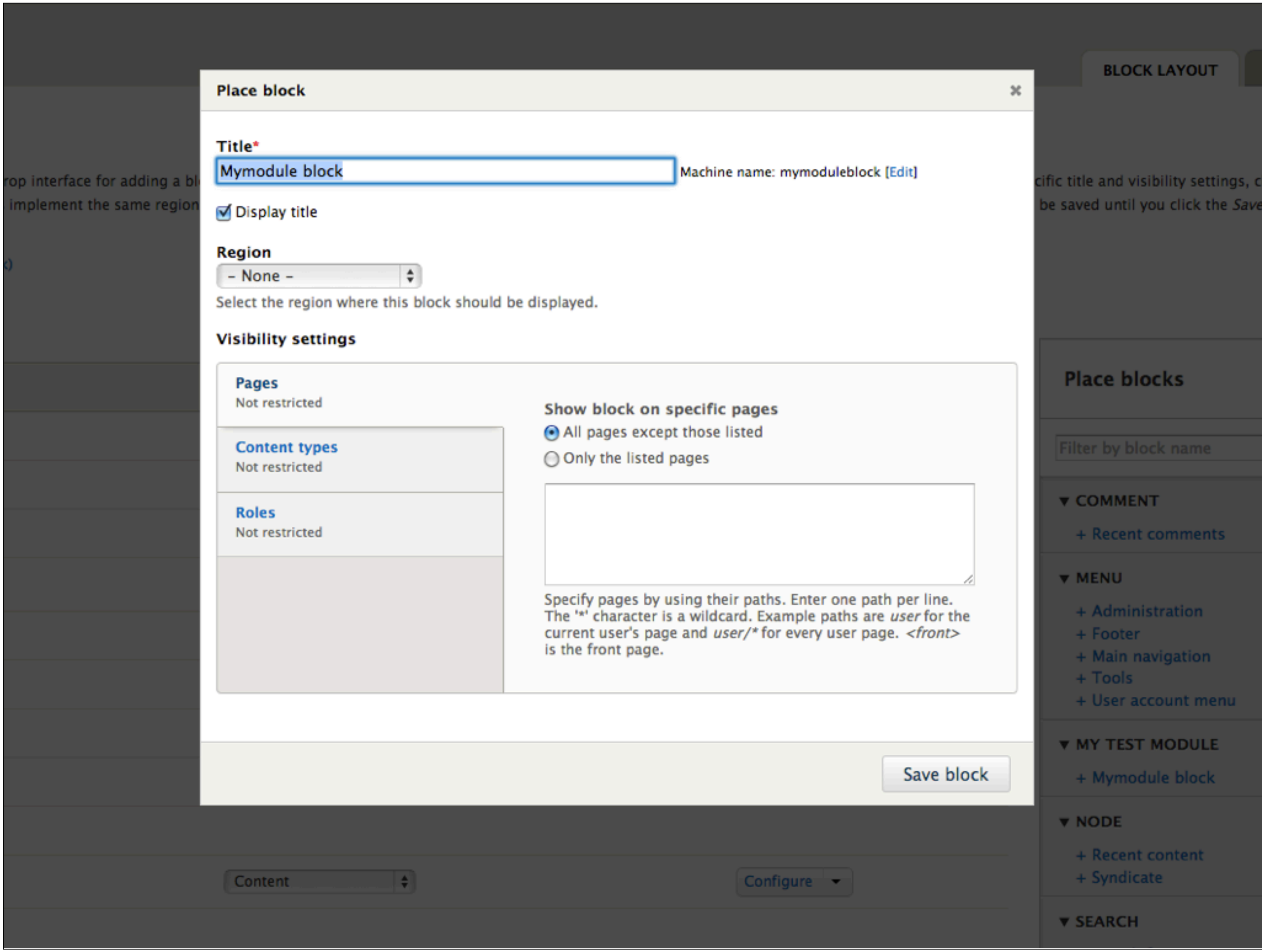
▼ MENU

+ Administration
+ Footer
+ Main navigation
+ Tools
+ User account menu

▼ MY TEST MODULE

+ Mymodule block

Blocks Admin Page Has a New Section: **Place blocks**



Place block

Title*

Mymodule block

Machine name: mymoduleblock [Edit]

[✓] Display title

Region

– None –

Select the region where this block should be displayed.

Visibility settings

Pages
Not restricted

Content types
Not restricted

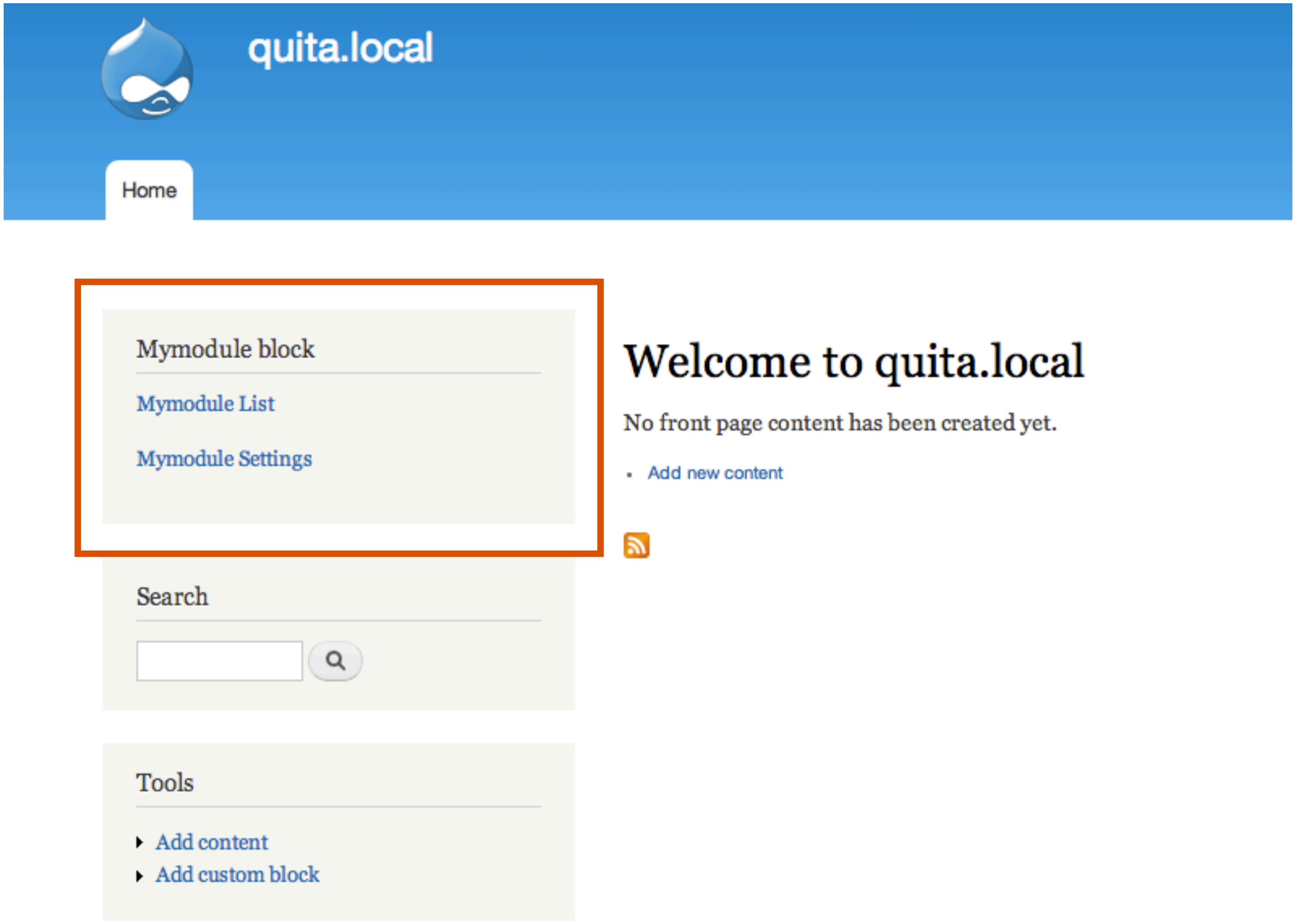
Roles
Not restricted

Show block on specific pages

[● All pages except those listed]
○ Only the listed pages

Specify pages by using their paths. Enter one path per line. The '*' character is a wildcard. Example paths are *user* for the current user's page and *user/** for every user page. *<front>* is the front page.

Save block



quita.local

Home

Mymodule block

Mymodule List

Mymodule Settings

Search

Tools

- Add content
- Add custom block

Welcome to quita.local

No front page content has been created yet.

- Add new content

Hook to Plugin Comparison:

Drupal 7.x	**Drupal 8.x**
`hook_block_info()`	`BlockManager::getDefinitions()`
`hook_block_view($delta)`	`BlockPluginInterface::build()`
?	`BlockPluginInterface::access()`
`hook_block_configure($delta)`	`BlockPluginInterface::blockForm($form, &$form_state)`
?	`BlockPluginInterface::blockValidate($form, &$form_state)`
`hook_block_save($delta, $edit)`	`BlockPluginInterface::blockSubmit($form, &$form_state)`

Block Discovery and Annotations

- Each Plugin type must be in the expected class namespace for your module - for blocks:
  `namespace Drupal\mymodule\Plugin\Block;`
- Most core plugins have a custom annotation class- you have to use the right one for your plugin.
- The annotation class provides both a documentation of the possible keys in the plugin definition and default values.
- There is a generic Plugin annotation class, but you should create a specific subclass for your plugin.

```
/**
 * Defines a Block annotation object.
 *
 * @Annotation
 */
class Block extends Plugin {

  /**
   * The plugin ID.
   *
   * @var string
   */
  public $id;

  /**
   * The administrative label of the block.
   *
   * @var \Drupal\Core\Annotation\Translation
   *
   * @ingroup plugin_translatable
   */
  public $admin_label;
}
```

Creating Your Own Plugins

- You want to upgrade your module to Drupal 8 and it defined an info hook or had a ctools plugin type.
- Annotation based discovery should be the default.
- It keeps the meta-data together with the class and it suited for most plugins where the actual class (code) is different for most plugins.
- The YAML discovery is good for a case like local tasks where the vast majority use a common class, but a few will implement a different one (e.g. to provide a dynamic title).

Plugin and General 8.x Resources

- Demo code used for this presentation: https://drupal.org/sandbox/pwolanin/2087657
- *Converting 7.x modules to 8.x* https://drupal.org/update/modules/7/8
- *Plugin API in Drupal 8* https://drupal.org/node/2087839
- *Understanding Drupal 8's plugin system* http://previousnext.com.au/blog/understanding-drupal-8s-plugin-system

Drupal 8 Features I Mentioned

- Widespread use of interfaces makes it easier to replace almost any implementation.
- Tabs are grouped regardless of path hierarchy.
- Route name rather than system path is unique.
- Multiple routes can serve the same path (HTML vs. JSON or GET vs. POST).
- “variables” split into config (deployable) and state.
- YAML as a standard for config and data files.
- Multiple instances of the same block.

To Sum It Up

- Plugins are a way to combine the discovery of available functionality with the actual implementation of the functionality.
- In Drupal 7, the combination of an info hook and multiple other hook functions (potentially in different files) served the same purpose.
- When defining your own plugins, use Annotation-based discovery unless you have a very clear reason for a different type.