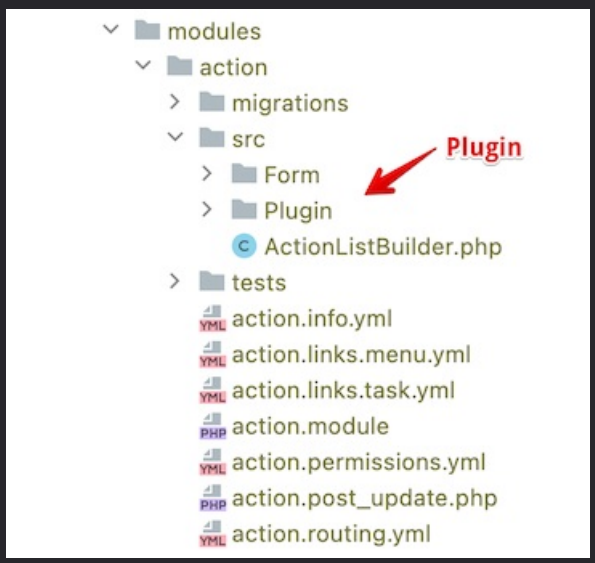
Blocks

TABLE OF CONTENTS

Blocks are plugins, which are reusable pieces of code following design patterns. Plugins are also used to define views arguments, field formatters, field widgets, etc. The source files for blocks are found in each module's /src/Plugin directory.



SEE ALSO

- Plugin API overview
- Annotations-based plugins

Create a block with Drush generate

Use Drush's code generation ability to quickly generate the code you need to create your own custom block.

First generate a module if you don't have one. Here we generate a module called Block Module with a machine name: block_module.

```
$ drush generate module
```

```
Welcome to module generator!

---------------------------------------------------------------

Module name \[Web\]:

➤ Block Module

Module machine name \[block_module\]:

➤

Module description \[Provides additional functionality for the site.\]:

➤ Custom module to explore Drupal blocks

Package \[Custom\]:

➤

Dependencies (comma separated):

➤

Would you like to create module file? \[No\]:

➤ yes

Would you like to create install file? \[No\]:

➤

Would you like to create libraries.yml file? \[No\]:

➤

Would you like to create permissions.yml file? \[No\]:

➤

Would you like to create event subscriber? \[No\]:

➤

Would you like to create block plugin? \[No\]:

➤

Would you like to create a controller? \[No\]:

➤

Would you like to create settings form? \[No\]:

➤

The following directories and files have been created or updated:
```

```
------------------------------------------------------------

 •
/Users/selwyn/Sites/ddev93/web/modules/custom/block_module/block_module.info.yml

 •
/Users/selwyn/Sites/ddev93/web/modules/custom/block_module/block_module.module
```

Use `drush generate` to create the code for a block. Specify the module name (e.g. block_module) so Drush knows where to put the block code. We also must give the block an admin label, plugin ID, and class.

```
$ drush generate block

Welcome to block generator!

------------------------------------------------------------

Module machine name \[web\]:

➤ block_module

Block admin label \[Example\]:

➤ Block Module Example

Plugin ID \[block_module_block_module_example\]:

➤

Plugin class \[BlockModuleExampleBlock\]:

➤

Block category \[Custom\]:

➤

Make the block configurable? \[No\]:

➤

Would you like to inject dependencies? \[No\]:

➤

Create access callback? \[No\]:

➤

The following directories and files have been created or updated:

------------------------------------------------------------

 •
/Users/selwyn/Sites/ddev93/web/modules/block_module/src/Plugin/Block/BlockModuleExampleBlock.php
```

This generates a file at `web/modules/custom/block_module/src/Plugin/Block/BlockModuleExampleBlock.php` that looks like this:

```php
<?php

namespace Drupal\block_module\Plugin\Block;

use Drupal\Core\Block\BlockBase;

/**
 * Provides a block module example block.
 *
 * @Block(
 *   id = "block_module_block_module_example",
 *   admin_label = @Translation("Block Module Example"),
 *   category = @Translation("Custom")
 * )
 */
class BlockModuleExampleBlock extends BlockBase {

  /**
   * {@inheritdoc}
   */
  public function build() {
    $build['content'] = [
      '#markup' => $this->t('It works!'),
    ];
    return $build;
  }

}
```

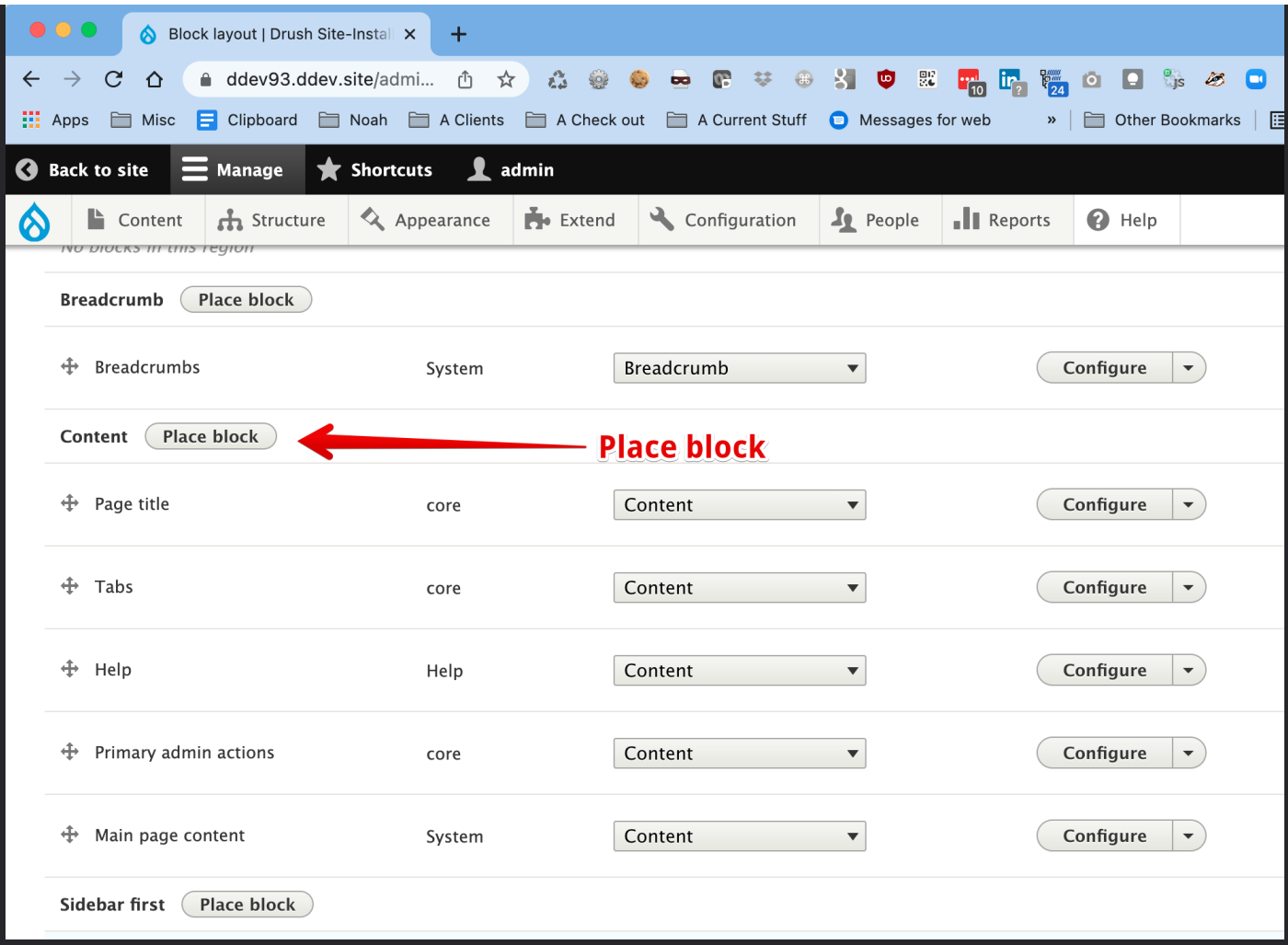
Enable the module with:

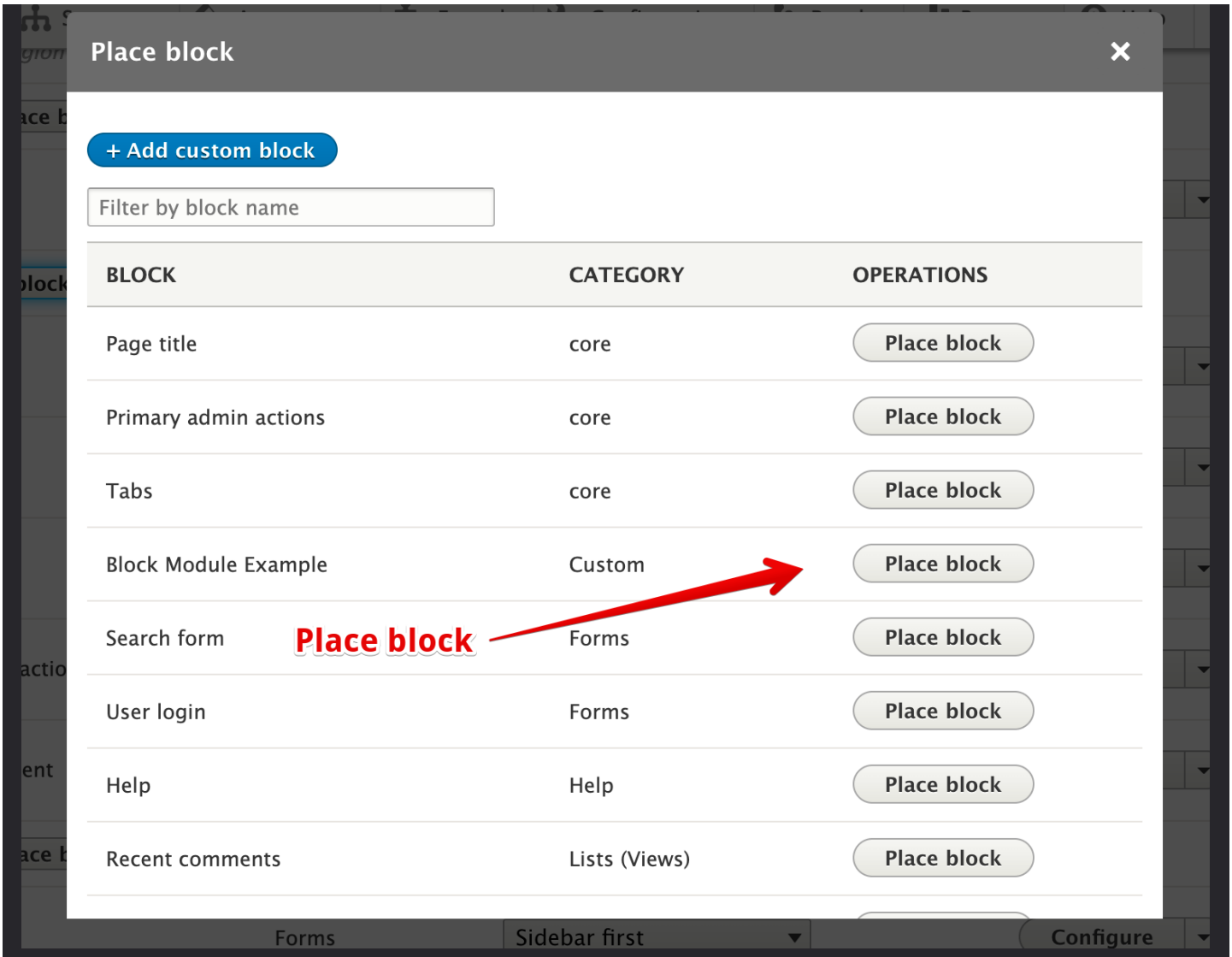
```
ddev drush en block_module
```

clear the cache with:

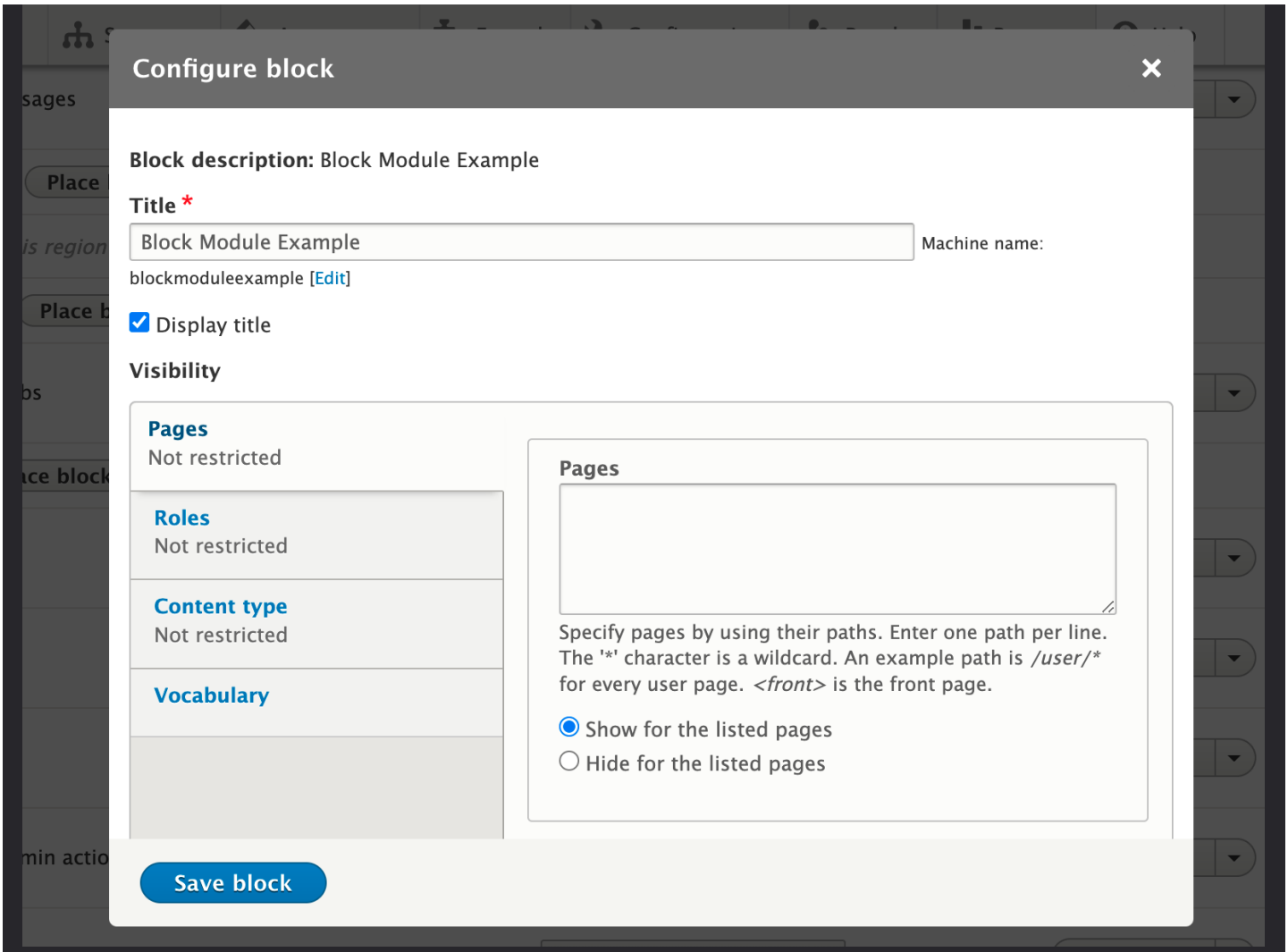
```
ddev drush cr
```

In Drupal, navigate to /admin/structure/block and place the block ("block module example") in the content area. See the diagram below on how to place the block in the content area.

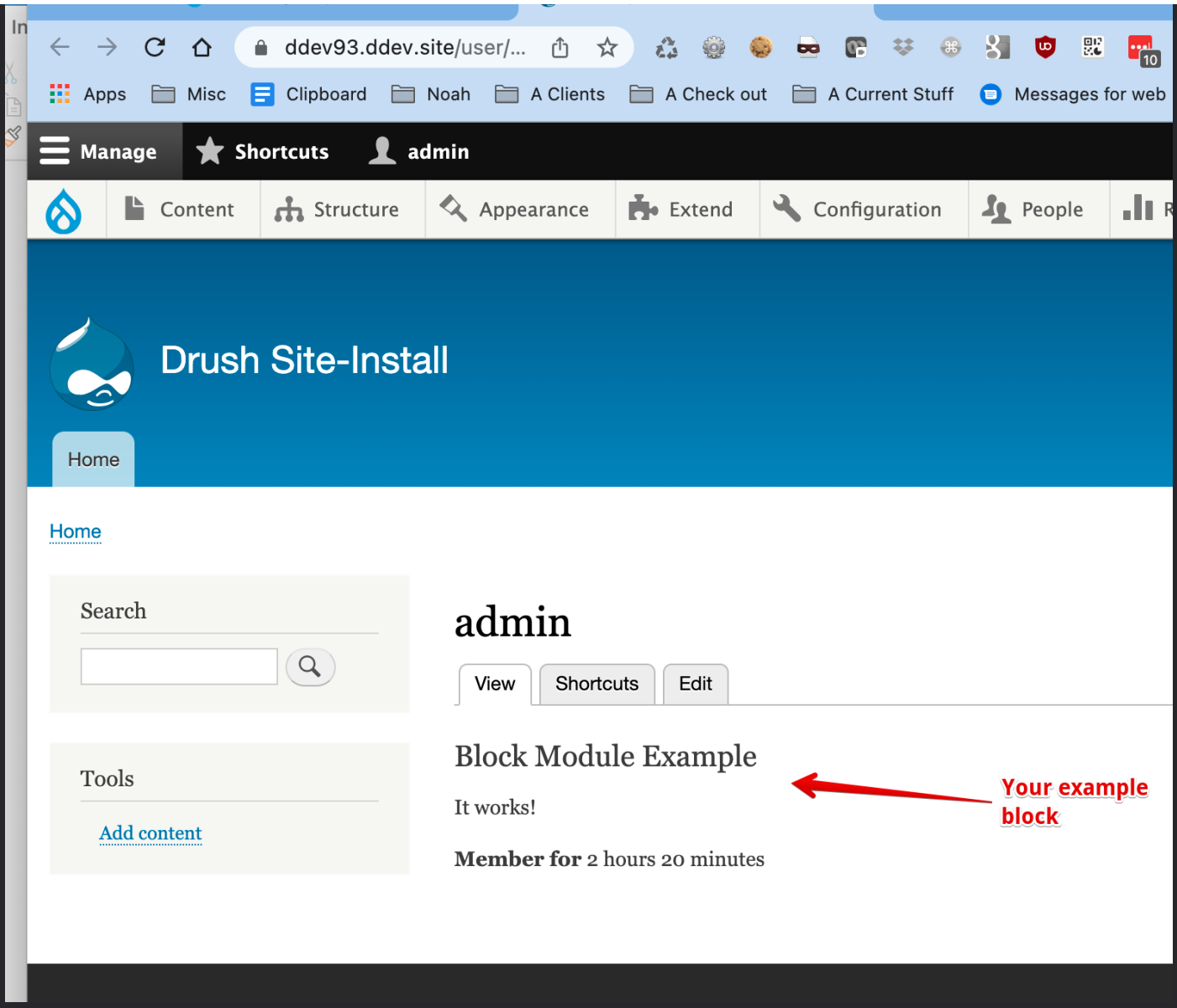

Block layout | Drush Site-Instal
ddev93.ddev.site/admi...
Apps
Misc
Clipboard
Noah
A Clients
A Check out
A Current Stuff
Messages for web
Other Bookmarks
Back to site
Manage
Shortcuts
admin
Content
Structure
Appearance
Extend
Configuration
People
Reports
Help
Breadcrumb
Place block
Breadcrumbs
System
Breadcrumb
Configure
Content
Place block
Place block
Page title
core
Content
Configure
Tabs
core
Content
Configure
Help
Help
Content
Configure
Primary admin actions
core
Content
Configure
Main page content
System
Content
Configure
Sidebar first
Place block



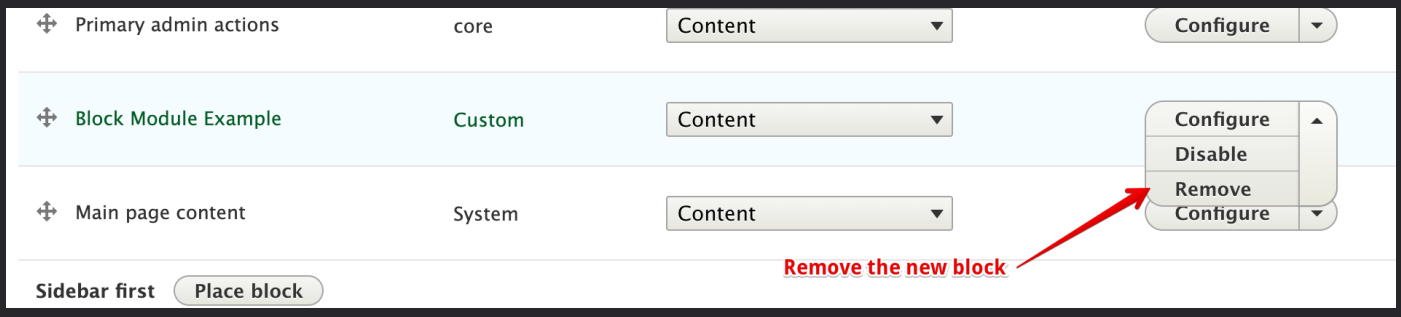
You may have to clear the Drupal cache again to get the new block to show up in the list. After clicking “Place block,” a “Configure block” screen appears. You can safely just click “Save block.”



Navigate back to the home page of the site and you'll see your block appearing. Screenshot below:



You can safely remove the block via the block layout page, choose “remove” from the dropdown next to your “Block Module Example”



Anatomy of a custom block with dependency injection

The block class PHP file is usually in \<Drupal web root\>/modules/custom/mymodule/src/Plugin/Block.

e.g. dev1/web/modules/custom/image_gallery/src/Plugin/Block/ImageGalleryBlock.php

or

dev1/web/modules/contrib/examples/block_example/src/Plugin/Block/ExampleConfigurableTextBlock.php

Specify namespace:

```
namespace Drupal\abc_wea\Plugin\Block;
```

Blocks always extend BlockBase but can also implement other interfaces… see below.

Class ImageGalleryBlock extends BlockBase

If you want to use Dependency Injection, implement: `ContainerFactoryPluginInterface`

e.g.

```
class ImageGalleryBlock extends BlockBase implements
ContainerFactoryPluginInterface {
```

Be sure to include:

```
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
```

And for annotation translation:

```
use Drupal\Core\Annotation\Translation;
```

You can annotate like this:

```
/**
 * Hello World Salutation block.
 *
 * @Block(
 *   id = "hello_world_salutation_block",
 *   admin_label = @Translation("Hello world salutation"),
 *   category = @Translation("Custom")
 * )
 */
```

Or like this:

```
/**
 * Provides an image gallery block.
 *
 * @Block(
 *   id = "ig_product_image_gallery",
 *   admin_label = @Translation("Product Image Gallery"),
 *   category = @Translation("Image Display"),
 *   context = {
 *     "node" = @ContextDefinition(
 *       "entity:node",
 *       label = @Translation("Current Node")
 *     )
 *   }
 * )
 */
```

In most cases you will implement ContainerFactoryPluginInterface. Plugins require this for dependency injection. So don't forget:

```
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;

class HelloWorldSalutationBlock extends BlockBase implements ContainerFactoryPluginInterface {
```

If you want dependency injection, you will need a create() function.

This will call the constructor (to do lazy loading) and call the container to `->get()` the service you need. In the example below `$container->get('hello_world.salutation')` does the trick. `return new static()` calls your class constructor.

Be sure to add your service to the list of parameters in the constructor: `$container->get('hello_world.salutation')`.

```
/**
 * {@inheritdoc}
 */
public static function create(ContainerInterface $container, array $configuration, $plugin_id, $plugin_definition) {
  return new static(
    $configuration,
    $plugin_id,
    $plugin_definition,
    $container->get('hello_world.salutation')
  );
}
```

Here are your __constructor() and a build() functions. See the 4th param – HelloWorldSalutationService $salutation – that's the injected service.

```
/**
 * Construct.
 *
 * @param array $configuration
 *   A configuration array containing information about the plugin instance.
 * @param string $plugin_id
 *   The plugin_id for the plugin instance.
 * @param string $plugin_definition
 *   The plugin implementation definition.
 * @param \Drupal\hello_world\HelloWorldSalutation $salutation
 */
public function __construct(array $configuration, $plugin_id, $plugin_definition, HelloWorldSalutationService $salutation) {
  parent::__construct($configuration, $plugin_id, $plugin_definition);
  $this->salutation = $salutation;
}
```

```
/**
 * {@inheritdoc}
 */
public function build() {
  return [
    '#markup' => $this->salutation->getSalutation(),
  ];
}
```

TODO: NEED A BETTER EXAMPLE OF A D.I. BLOCK HERE especially showing a build()

Create a block with an entityQuery

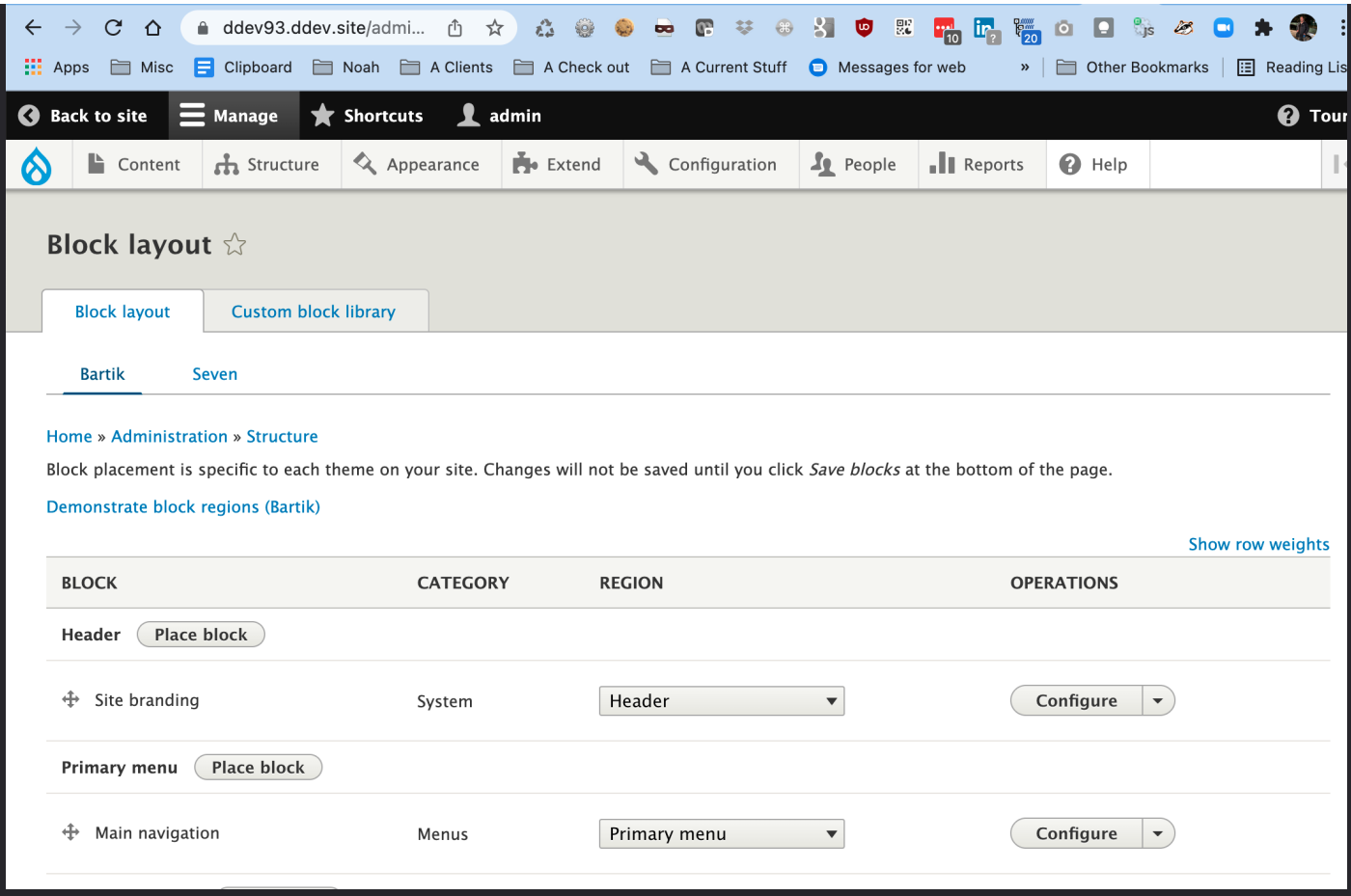
You often need to query some data from Drupal and display it in a block.

Here is a simple block that loads all published content of type "page" and renders the titles. You could sort them by creation date by adding this to the $query variable: ->sort('created' , 'DESC');

```php
namespace Drupal\opinions_module\Plugin\Block;

use Drupal\Core\Block\BlockBase;
use Drupal\Core\Annotation\Translation;

/**
 * Provides OpinionLanding Block.
 *
 * @Block(
 *   id = "opinion_landing",
 *   admin_label = @Translation("Opinion landing block"),
 *   )
 *
 * @package Drupal\oag_opinions\Plugin\Block
 */
class OpinionLanding extends BlockBase {

  public function build() {

    $entity_type = 'node';
    $storage = \Drupal::entityTypeManager()->getStorage($entity_type);
    $query = \Drupal::entityQuery('node')
      ->condition('type', 'page')
      ->condition('status', 1) ;
    $nids = $query->execute();
    $nodes = $storage->loadMultiple($nids);

    $render_array = [];
    foreach ($nodes as $node) {
      $render_array[] = [
        '#type' => 'markup',
        '#markup' => '<p>' . $node->getTitle(),
      ];
    }

    return $render_array;
```

Create a Block with a corresponding config form

Here is an example which includes a block and a corresponding config form that controls what is displayed in the block. The block can be placed using the Block Layout system in Drupal at /admin/structure/block (shown below) or via twig in a template file.



The config form definition

The config form is defined in docroot/modules/custom/quick_pivot/src/Form/QuickPivotConfigForm.php with a class which extends ConfigFormBase because this form is there for configuring its block:

```
class QuickPivotConfigForm extends ConfigFormBase {
```

In the class are the getFormId(), getEditableConfigName(), buildForm() and submitForm() functions which are all pretty straightforward.

The routing.yml file

Then in docroot/modules/custom/quick_pivot/quick_pivot.routing.yml we specify the route where we invoke the form.

Besides the quick_pivot.info.yml (module info) file, that should be all you need to make the config for the block.

The Block definition

Now for the block that users see (also the one that pops up in the block configuration) in docroot/modules/custom/quick_pivot/src/Plugin/Block/QuickPivotSubscribeBlock.php

We define the block with its annotation:

```
/**
 * Provides a cart block.
 *
 * @Block(
 *   id = "quick_pivot_subscribe_block",
 *   admin_label = @Translation("QuickPivot Subscribe Block"),
 *   category = @Translation("QuickPivot Subscribe")
 * )
 */
class QuickPivotSubscribeBlock extends BlockBase implements ContainerFactoryPluginInterface {
```

It implements ContainerFactoryPluginInterface to allow dependency injection. This is critical for plugins or blocks. More at https://chromatichq.com/blog/dependency-injection-drupal-8-plugins. All this interface defines is the create() method. Because we are using dependency injection, we need both a create() and a __constructor().

Here is the create()

```
public static function create(ContainerInterface $container, array $configuration, $plugin_id, $plugin_definition) {
  return new static(
    $configuration,
    $plugin_id,
    $plugin_definition,
    $container->get('config.factory'),
    $container->get('form_builder')
  );
}
```

Here is the constructor:

```
public function __construct(array $configuration, $plugin_id, $plugin_definition, ConfigFactoryInterface $config_factory, FormBuilderInterface $form_builder) {
  parent::__construct($configuration, $plugin_id, $plugin_definition);

  $this->configFactory = $config_factory;
  $this->formBuilder = $form_builder;
}
```

And finally the build() method:

```
public function build() {
  return $this->formBuilder->getForm('Drupal\quick_pivot\Form\QuickPivotSubscribeForm');
}
```

Here is the docroot/modules/custom/quick_pivot/src/Form/QuickPivotSubscribeForm.php:

```
<?php

namespace Drupal\quick_pivot\Form;

use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Form\FormBuilderInterface;
use Drupal\Core\Ajax\AjaxResponse;
use Drupal\Core\Ajax\ReplaceCommand;
use Drupal\Core\Ajax\CssCommand;
use Drupal\Core\Ajax\HtmlCommand;
use Drupal\Core\Ajax\AppendCommand;
use Drupal\quick_pivot\QuickPivotApiInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Provides a form for users to subscribe to QuickPivot.
 */
class QuickPivotSubscribeForm extends FormBase {

  /**
   * {@inheritdoc}
   */
  public function getFormId() {
    return 'quick_pivot_subscribe_form';
  }

  /**
   * {@inheritdoc}
```

```
  */
  public function buildForm(array $form, FormStateInterface $form_state) {
    $form['#id'] = 'quick-pivot-subscribe-form';
    $form['#cache'] = ['max-age' => 0];
    $form['#attributes'] = ['autocomplete' => 'off'];

    $form['email'] = [
      '#type' => 'textfield',
      '#id' => 'quick-pivot-email',
      '#placeholder' => $this->t('Email address'),
      '#attributes' => ['class' => ['edit-quick-pivot-email']],
      '#prefix' => '<div class="subscriber-email-msg">',
      '#suffix' => '</div>',
    ];
    $form['actions']['subscribe_submit'] = [
      '#type' => 'submit',
      '#value' => $this->t('Sign Up'),
      '#name' => 'quick_pivot_subscribe_form_submit_button',
      '#ajax' => [
        'callback' => 'Drupal\quick_pivot\Form\QuickPivotSubscribeForm::quickPivotAjaxSubmit',
        'wrapper' => 'quick-pivot-subscribe-form',
        'progress' => ['type' => 'throbber', 'message' => NULL],
      ],
    ];
    $form['message'] = [
      '#type' => 'markup',
      '#markup' => '<div id="quick-pivot-message-area"></div>',
    ];

    return $form;
  }

  /**
   * {@inheritdoc}
   */
  public function validateForm(array &$form, FormStateInterface $form_state) {

  }

  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state) {

  }

  /**
   * {@inheritdoc}
   */
  public static function quickPivotAjaxSubmit(array &$form, FormStateInterface $form_state) {
    $validate = TRUE;
    $email = trim($form_state->getValue('email'));
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
      $message = t('Please enter a valid email address.');
      $validate = FALSE;
      $css_border = ['border' => '1px solid red'];
      $css_color = ['color' => 'red'];
    }
    if ($validate) {
      $css_border = ['border' => '1px solid green'];
```

```
    $css_border = ['border' => '1px solid green'];
    $css_color = ['color' => 'green'];
    $response = \Drupal::service('quick_pivot.api')->subscribeEmail($email);
    if (strpos(reset($response), 'Success') !== FALSE) {
      $message = t('Thank you for signing up. Your subscription has been activated.');
    }
    else {
      $message = t('Your subscription could not be processed.');
    }
  }

  $response = new AjaxResponse();

  $quick_pivot_form = \Drupal::formBuilder()->rebuildForm('quick_pivot_subscribe_form', $form_state);
  if ($validate) {
    $quick_pivot_form['email']['#value'] = '';
    $quick_pivot_form['email']['#placeholder'] = t('Email address');
  }
  $response->addCommand(new ReplaceCommand('#quick-pivot-subscribe-form', $quick_pivot_form));
  $response->addCommand(new CssCommand('#edit-quick-pivot-email', $css_border));
  $response->addCommand(new HtmlCommand('#quick-pivot-message-area', $message));
  $response->addCommand(new CssCommand('#quick-pivot-message-area', $css_color));
  return $response;
 }
}
```

Here is the entire QuickPivotConfigForm.php file:

```
<?php

namespace Drupal\quick_pivot\Form;

use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Form\ConfigFormBase;

/**
 * Configure Websphere settings for this site.
 */
class QuickPivotConfigForm extends ConfigFormBase {

  /**
   * {@inheritdoc}
   */
  public function getFormId() {
    return 'quick_pivot_settings';
  }

  /**
   * {@inheritdoc}
   */
  protected function getEditableConfigNames() {
    return ['quick_pivot.settings'];
  }

  /**
   * {@inheritdoc}
   */
  public function buildForm(array $form, FormStateInterface $form_state) {
```

```
    $config = $this->config('quick_pivot.settings');

    $form['quick_pivot_settings'] = [
      '#type' => 'details',
      '#title' => $this->t('Quick Pivot API Settings'),
      '#open' => TRUE,
      '#weight' => 1,
    ];

    $form['quick_pivot_settings']['api_end_point'] = [
      '#type' => 'textfield',
      '#title' => $this->t('API End point'),
      '#description' => $this->t("Enter the API end point URL."),
      '#default_value' => $config->get('quick_pivot_settings.api_end_point'),
      '#required' => TRUE,
      '#size' => 100,
    ];

    $form['quick_pivot_settings']['user_guid'] = [
      '#type' => 'textfield',
      '#title' => $this->t('User GUID'),
      '#description' => $this->t("SOAP API User GUID"),
      '#default_value' => $config->get('quick_pivot_settings.user_guid'),
      '#required' => TRUE,
      '#size' => 100,
    ];

    $form['quick_pivot_settings']['account'] = [
      '#type' => 'textfield',
      '#title' => $this->t('Account'),
      '#description' => $this->t("SOAP API Account"),
      '#default_value' => $config->get('quick_pivot_settings.account'),
      '#required' => TRUE,
      '#size' => 100,
    ];

    $form['quick_pivot_settings']['sender'] = [
      '#type' => 'textfield',
      '#title' => $this->t('Sender'),
      '#description' => $this->t("SOAP API Sender"),
      '#default_value' => $config->get('quick_pivot_settings.sender'),
      '#required' => TRUE,
      '#size' => 100,
    ];

    return parent::buildForm($form, $form_state);
  }

  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state) {

    $this->config('quick_pivot.settings')
        ->set('quick_pivot_settings.api_end_point', $form_state->getValue('api_end_point'))
        ->set('quick_pivot_settings.user_guid', $form_state->getValue('user_guid'))
        ->set('quick_pivot_settings.account', $form_state->getValue('account'))
        ->set('quick_pivot_settings.sender', $form_state->getValue('sender'))
        ->save();
```

```
    parent::submitForm($form, $form_state);
  }

}
```

And the QuickPivotSubscribeBlock.php:

```
<?php

namespace Drupal\quick_pivot\Plugin\Block;

use Drupal\Core\Block\BlockBase;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Drupal\Core\Config\ConfigFactoryInterface;
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
use Drupal\Core\Form\FormBuilderInterface;

/**
 * Provides a cart block.
 *
 * @Block(
 *   id = "quick_pivot_subscribe_block",
 *   admin_label = @Translation("QuickPivot Subscribe Block"),
 *   category = @Translation("QuickPivot Subscribe")
 * )
 */
class QuickPivotSubscribeBlock extends BlockBase implements ContainerFactoryPluginInterface {

  /**
   * The configuration factory.
   *
   * @var \Drupal\Core\Config\ConfigFactoryInterface
   */
  protected $configFactory;

  /**
   * The form builder.
   *
   * @var \Drupal\Core\Form\FormBuilderInterface
   */
  protected $formBuilder;

  /**
   * Constructor for the QuickPivot subscribe block.
   *
   * @param array $configuration
   *   The block configuration.
   * @param string $plugin_id
   *   The plugin_id for the plugin instance.
   * @param mixed $plugin_definition
   *   The plugin implementation definition.
   * @param \Drupal\Core\Config\ConfigFactoryInterface $config_factory
   *   The configuration factory.
   * @param \Drupal\Core\Form\FormBuilderInterface $form_builder
   *   The form builder.
   */
  public function __construct(array $configuration, $plugin_id, $plugin_definition, ConfigFactoryInterface $config_factory, FormBuilderInterface $form_builder) {
    parent::__construct($configuration, $plugin_id, $plugin_definition);
```

```
    $this->configFactory = $config_factory;
    $this->formBuilder = $form_builder;
  }

  /**
   * {@inheritdoc}
   */
  public static function create(ContainerInterface $container, array $configuration, $plugin_id, $plugin_definition) {
    return new static(
      $configuration,
      $plugin_id,
      $plugin_definition,
      $container->get('config.factory'),
      $container->get('form_builder')
    );
  }

  /**
   * Builds the cart block.
   *
   * @return array
   *   A render array.
   */
  public function build() {
    return $this->formBuilder->getForm('Drupal\quick_pivot\Form\QuickPivotSubscribeForm');
  }

}
```

And here is the routing file: docroot/modules/custom/quick_pivot/quick_pivot.routing.yml

```
quick_pivot.config:
  path: '/admin/config/quick_pivot/settings'
  defaults:
    _form: 'Drupal\quick_pivot\Form\QuickPivotConfigForm'
    _title: 'Quick Pivot Settings'
  requirements:
    _permission: 'administer site configuration'
```

And for the icing, We also specify a menu item so users can access the configuration form via the menu system at docroot/modules/custom/quick_pivot/quick_pivot.links.menu.yml.

```
quick_pivot.config:
  title: 'QuickPivot API settings'
  description: 'Configure the QuickPivot API Settings.'
  parent: system.admin_config_services
  route_name: quick_pivot.config
  weight: 1
```

Modify a block with hook_block_view_alter or hook_block_build_alter

Some drupal hooks only run inside a contributed modules and some only inside a theme and some both.

```
function themename_preprocess_block(&$variables)
{
    if ($variables['plugin_id'] == 'entity_browser_block:department_info') {
        $variables['#attached']['library'][] = 'drupal/libraryname';
    }
```

What's described below could potentially be done on a theme preprocess for the block.

If you need to modify a block, you can supposedly use `hook_block_view_alter` or `hook_block_build_alter`, although I haven't been able to make this work… hmm.

There is a comment that may be worth exploring at https://api.drupal.org/api/drupal/core%21modules%21block%21block.api.php/function/hook_block_view_alter/8.2.x.

To alter the block content you must add a `#pre_render` in the `hook_block_view_alter` hook.

In https://drupal.stackexchange.com/a/215948 there is an example which fills in the `$build['#pre_render'][]` array with a string.

In an example on that stackexchange site, this function is provided:

```
function yourmodule_block_view_alter(array &$build, \Drupal\Core\Block\BlockPluginInterface $block) {
  if ($block->getBaseId() === 'system_powered_by_block') {
    $build['#pre_render'][] = '_yourmodule_block_poweredby_prerender';
  }
```

I think this is the version I tried:

```
/**
 * Implements hook_block_build_alter().
 */
function plug_academy_core_block_build_alter(array &$build, \Drupal\Core\Block\BlockPluginInterface $block) {
  if ($block->getPluginId() == 'system_menu_block:account') {
    $build['#cache']['contexts'][] = 'url';
  }
//  else if ($block->getBaseId() === 'block_content') {
//    if ($block->label() === "Home Page Alert") {
//      $build['content'] = '<p>New content built here!</p>';
//
//    }
//  }
}
```

And I discovered an example from a project where the `$build['#pre_render'][]` array is populated with a function. I'm not sure what that function did – presumably returned some text to be rendered.

```
/**
 * Implements hook_block_view_alter().
 */
function pega_academy_core_block_view_alter(array &$build, \Drupal\Core\Block\BlockPluginInterface $block) {
  if ($block->getBaseId() === 'block_content') {
    if ($block->label() === "Home Page Alert") {
      $build['#pre_render'][] = 'Drupal\pega_academy_core\Controller\DashboardController::home_page_alert_prerender';
//      $build['content'] = '<p>New content built here!</p>';

    }
  }
}
```

Disable caching in a block

From `docroot/modules/custom/websphere_commerce/modules/cart/src/Plugin/Block/CartSummary.php`:

```
/**
 * {@inheritdoc}
 */
public function getCacheMaxAge() {
  return 0;
}
```

Add a configuration form to your block

Making a block configurable means it has a form where you can specify its settings, e.g., the configuration form for the menu block module allows you to specify menu levels. Ignore this if your block does not need any configuration.

To make your block configurable, override 3 methods from BlockBase.

1. defaultConfiguration
2. blockForm
3. blockSubmit

Here `defaultConfiguration()` returns a block_count of 5.

```
/**
 * {@inheritdoc}
 */
public function defaultConfiguration() {
  // By default, the block will display 5 thumbnails.
  return [
    'block_count' => 5,
  ];
}
```

`blockForm()` is used to create a configuration form:

```
/**
 * {@inheritdoc}
 */
public function blockForm($form, FormStateInterface $form_state) {
  $range = range(2, 20);
  $form['block_count'] = [
    '#type' => 'select',
    '#title' => $this->t('Number of product images in block'),
    '#default_value' => $this->configuration['block_count'],
    '#options' => array_combine($range, $range),
  ];
  return $form;
}
```

And `blockSubmit()` handles the submission of the config form. You don't need to save anything. The data is saved automatically into the Drupal config system. You just specify a configuration key like `$this->configuration['block_count']` and the rest is handled for you.

```
/**
 * {@inheritdoc}
 */
public function blockSubmit($form, FormStateInterface $form_state) {
  $this->configuration['block_count'] = $form_state->getValue('block_count');
}
```

The `build()` method does all the work of building a render array to display whatever your block wants to display. Here is an example of a

build() function.

```
/**
 * {@inheritdoc}
 */
public function build() {
  $build = [];
  $node = $this->getContextValue('node');

  // Determine if we are on a page that points to a product.
  $product = $this->getProduct($node);
  if ($product) {

    // Retrieve the product images
    $image_data = $this->productManagerService->retrieveProductImages($product);
    $block_count = $this->configuration['block_count'];
    $item_count = 0;
    $build['list'] = [
      '#theme' => 'item_list',
      '#items' => [],
    ];

    $build['list']['#items'][0] = [
      '#type' => 'markup',
      '#markup' => $this->t('There were no product images to display.')
    ];

    while ($item_count < $block_count && isset($image_data[$item_count])) {
      $file = File::load($image_data[$item_count]['target_id']);
      $link_text = [
        '#theme' => 'image_style',
        '#uri' => $file->getFileUri(),
        '#style_name' => 'product_thumbnail',
        '#alt' => $image_data[$item_count]['alt'],
      ];

      // Modal dialog
      // see https://www.drupal.org/node/2488192 for more on modals
      $options = [
        'attributes' => [
          'class' => [
            'use-ajax',
          ],
          'data-dialog-type' => 'modal',
          'data-dialog-options' => Json::encode([
            'width' => 700,
          ]),
        ],
      ];
      $url = Url::fromRoute('abc_prg.display_product_image', ['node' => $product->nid->value, 'delta' => $item_count]);
      $url->setOptions($options);
      $build['list']['#items'][$item_count] = [
        '#type' => 'markup',
        '#markup' => Link::fromTextAndUrl($link_text, $url)
          ->toString(),
      ];
      $item_count++;
    }
    $build['#attached']['library'][] = 'core/drupal.dialog.ajax';
  }
```

```
  }
  return $build;
}
```

One last item. Configuration expects a schema for things being saved. Here we create a .schema.yml in /config/schema and it looks like:

```
# Schema for the configuration files for my module.

block.settings.alchemy_block:
  type: block_settings
  label: 'Alchemy block'
  mapping:
    block_count:
      type: integer
      label: 'Block count'
```

Block display not updating after changing block content

From [Nedcamp video on caching by Kelly Lucas, November 2018]

In a twig template, if you just want to render one or more fields (instead of the entire node), Drupal may not be aware if the content has changed, and will sometimes show old cached content. To resolve this, define a view mode and call `content | render` and assign the result to a variable like this:

```
set blah = content|render
```

Be sure to surround the above code with curly brace and percentage sign delimeters. Unfortunately these don't always render correctly in this document so I've had to remove them for now.

Adding this render call will cause Drupal to render the content for that node, which will cause a check of the caches and make sure the most current content is rendered.

Then add your fields:

```
  {content.field_one}  etc.
```

Block Permission (blockAccess)

This code is taken from the Drupal core user_login_block (UserLoginBlock.php). It allows access to the block if the user is logged out and is not on the login or logout page. The access is cached based on the current route name and the user's current role being anonymous. If these are not passed, the access returned is forbidden and the block is not built.

```
use Drupal\Core\Access\AccessResult;

$account = \Drupal::currentUser();



/**
 * {@inheritdoc}
 */
protected function blockAccess(AccountInterface $account) {
  $route_name = $this->routeMatch->getRouteName();
  if ($account->isAnonymous() && !in_array($route_name, ['user.login', 'user.logout'])) {
    return AccessResult::allowed()
      ->addCacheContexts(['route.name', 'user.roles:anonymous']);
  }
  return AccessResult::forbidden();
}
```

Another example from the Drupal core Copyright.php file:

```
// $account comes from
$account = \Drupal::currentUser();

//Get the route.
$route_name = \Drupal::routeMatch()->getRouteName();

// not on the user login and logout pages
if (!in_array($route_name, ['user.login', 'user.logout'])) {
  return AccessResult::allowed();
}

//Authenticated user
if ($account->isAuthenticated()) {
  return AccessResult::allowed();
}
//Anonymous user.
if ($account->isAnonymous()) {
  return AccessResult::forbidden();
}
```

Blocks shouldn’t talk to the router, NodeRouteContext and friends should

While it is possible for blocks to talk to the router, you can’t always count that they will be on a meaningful route i.e. are they being displayed on a node? So we should use context definition in the block annotation like this:

```
/**
 * Provides a 'Node Context Test' block.
 *
 * @Block(
 *   id = "node_block_test_context",
 *   label = @Translation("Node Context Test"),
 *   context_definitions = {
 *     "node" = @ContextDefinition("entity:node", label = @Translation("Node"))
 *   }
 * )
 */
```

This causes the block to be available only on various node pages (view, edit etc.). This can be changed:

```
 *   context_definitions = {
 *     "node" = @ContextDefinition("entity:node", label = @Translation("Node"),
 *       required = FALSE)
 *   }
```

The order of named options passed to ContextDefinition after the first argument does not matter.

Then in the block we check to make sure the user is viewing a node and that the user has `view rsvplist` permission. See the code below:

```
  protected function blockAccess(AccountInterface $account) {
    /** @var \Drupal\Core\Plugin\Context\Context $node */
    $cacheContext = $this->getContext('node');
    /** @var \Drupal\Core\Entity\Plugin\DataType\EntityAdapter $data */
    $data = $cacheContext->getContextData();
    /** @var \Drupal\node\NodeInterface $node */
    $node = $data->getValue();
    if ($node) {
      $nid = $node->id();
      if (is_numeric($nid)) {
        // See rsvp.permissions.yml for the permission string.
        return AccessResult::allowedIfHasPermission($account, 'view rsvplist');
      }
    }
    return AccessResult::forbidden();
  }
```

More at https://drupal.stackexchange.com/questions/145823/how-do-i-get-the-current-node-id/314152#314152

Note. While this practice is not recommended, the RSVP module does have an example of a block talking to the router i.e. `\Drupal::routeMatch()` - see https://git.drupalcode.org/project/rsvp_module/-/blob/1.0.x/src/Plugin/Block/RSVPBlock.php where the `blockAccess()` function grabs the `node` parameter and acts on it.

```
  /**
   * {@inheritdoc}
   */
  public function blockAccess(AccountInterface $account) {
    /** @var \Drupal\node\Entity\Node $node */
    $node = \Drupal::routeMatch()->getParameter('node');
    $nid = $node->nid->value;
    /** @var \Drupal\rsvp_module\EnablerService $enabler */
    $enabler = \Drupal::service('rsvp_module.enabler');
    if(is_numeric($nid)) {
      if($enabler->isEnabled($node)) {
        return AccessResult::allowedIfHasPermission($account, 'view rsvp_module');
      }
    }
    return AccessResult::forbidden();
  }
```

Values returned by blockAccess()

Some options that can be returned from blockAccess() are:

```
return AccessResult::forbidden();
return AccessResult::allowed();
return AccessResult::allowedIf(TRUE);
```

Back to top

Page last modified: Apr 13 2023.

Edit this page on GitHub