Render Arrays

TABLE OF CONTENTS

- Overview
- Overview of the Theme system and Render API.
- Caching
- Properties
- Image
- Simple Text
- Text with variable substitution (Placeholders)
- Wrap an element with a div with a class
- Prefix and suffix
- Date
- Image
- Several Url's.
- Two paragraphs
- A button that opens a modal dialog
- A link
- A link with a class
- A link and its TWIG template
- A link with parameters and a template file
- Simple unordered list
- Unordered list of links for a menu
- Nested Unordered List
- Select (dropdown)
- Select (dropdown) Ajaxified
- Limit allowed tags in markup
- Disable an element
- Resources

Overview

Render Arrays are the building blocks of a Drupal page. A render array is an associative array which conforms to the standards and data structures used in Drupal's Render API. The Render API is also integrated with the Theme API.

In many cases, the data used to build a page (and all parts of it) is kept as structured arrays until the final stage of generating a response. This provides enormous flexibility in extending, slightly altering or completely overriding parts of the page.

Render arrays are nested and thus form a tree. Consider them Drupal's "render tree" — Drupal's equivalent of the DOM.

Note: While render arrays and arrays used by the Form API share elements, properties and structure, many properties on form elements only have meaning for the Form API, not for the Render API. Form API arrays are transformed into render arrays by FormBuilder. Passing an unprocessed Form API array to the Render API may yield unexpected results.

Here is a simple render array that displays some text.

```
$my_render_array['some_item'] = [
  '#type' => markup,
  '#markup' => "This is a test",
];
```

All forms are Render arrays. This is important when need to use the form API to create forms.

This is mostly from https://www.drupal.org/docs/drupal-apis/render-api/render-arrays

Overview of the Theme system and Render API.

The main purpose of Drupal's Theme system is to give themes complete control over the appearance of the site, which includes the markup returned from HTTP requests and the CSS files used to style that markup. In order to ensure that a theme can completely customize the markup, module developers should avoid directly writing HTML markup for pages, blocks, and other user-visible output in their modules, and instead return structured "render arrays". Doing this also increases usability, by ensuring that the markup used for similar functionality on different areas of the site is the same, which gives users fewer user interface patterns to learn.

From the Render API overview at
https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/group/theme_render/10.0.x

Caching

You can specify caching information when creating render arrays. Cache keys, cache contexts, cache tags and cache max-age can all be defined.

The Drupal rendering process has the ability to cache rendered output at any level in a render array hierarchy. This allows expensive calculations to be done infrequently, and speeds up page loading. See the Cache API topic for general information about the cache system.

In order to make caching possible, the following information needs to be present:

- **Cache keys**: Identifiers for cacheable portions of render arrays. These should be created and added for portions of a render array that involve expensive calculations in the rendering process.
- **Cache contexts**: Contexts that may affect rendering, such as user role and language. When no context is specified, it means that the render array does not vary by any context.
- **Cache tags**: Tags for data that rendering depends on, such as for individual nodes or user accounts, so that when these change the cache can be automatically invalidated. If the data consists of entities, you can use \Drupal\Core\Entity\EntityInterface::getCacheTags() to generate appropriate tags; configuration objects have a similar method.
- **Cache max-age**: The maximum duration for which a render array maybe cached. Defaults to \Drupal\Core\Cache\Cache::PERMANENT (permanently cacheable).

Cache information is provided in the #cache property in a render array. In this property, always supply the cache contexts, tags, and max-age if a render array varies by context, depends on some modifiable data, or depends on information that's only valid for a limited time, respectively. Cache keys should only be set on the portions of a render array that should be cached. Contexts are automatically replaced with the value for the current request (e.g. the current language) and combined with the keys to form a cache ID. The cache contexts, tags, and max-age will be propagated up the render array hierarchy to determine cacheability for containing render array sections.

Here's an example of what a #cache property might contain:

```
'#cache' => [
  'keys' => ['entity_view', 'node', $node->id()],
  'contexts' => ['languages'],
  'tags' => $node->getCacheTags(),
  'max-age' => Cache::PERMANENT,
],
```

At the response level, you'll see `X-Drupal-Cache-Contexts` and `X-Drupal-Cache-Tags` headers.

Reproduced from the Render API overview at
https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/group/theme_render/10.0.x

Properties

Elements that start with # are properties and can include the following: `#type`, `#theme`, `#markup`, `#prefix`, `#suffix`, `#plain_text` or `#allowed_tags`.

Render arrays (at any level of the hierarchy) will usually have one of the following properties defined:

- **#type**: Specifies that the array contains data and options for a particular type of "render element" (for example, 'form', for an HTML form;

'textfield', 'submit', for HTML form element types; 'table', for a table with rows, columns, and headers). See Render elements below for more on render element types.

- **#theme**: Specifies that the array contains data to be themed by a particular theme hook. Modules define theme hooks by implementing hook_theme(), which specifies the input "variables" used to provide data and options; if a hook_theme() implementation specifies variable 'foo', then in a render array, you would provide this data using property '#foo'. Modules implementing hook_theme() also need to provide a default implementation for each of their theme hooks, normally in a Twig file. For more information and to discover available theme hooks, see the documentation of hook_theme() and the Default theme implementations topic.
- **#markup**: Specifies that the array provides HTML markup directly. Unless the markup is very simple, such as an explanation in a paragraph tag, it is normally preferable to use #theme or #type instead, so that the theme can customize the markup. Note that the value is passed through \Drupal\Component\Utility\Xss::filterAdmin(), which strips known XSS vectors while allowing a permissive list of HTML tags that are not XSS vectors. (For example, <script> and <style> are not allowed.) See \Drupal\Component\Utility\Xss::$adminTags for the list of allowed tags. If your markup needs any of the tags not in this list, then you can implement a theme hook and/or an asset library. Alternatively, you can use the key #allowed_tags to alter which tags are filtered.
- **#plain_text**: Specifies that the array provides text that needs to be escaped. This value takes precedence over #markup.
- **#allowed_tags**: If #markup is supplied, this can be used to change which tags are allowed in the markup. The value is an array of tags that Xss::filter() would accept. If #plain_text is set, this value is ignored.

Usage example:

```
$output['admin_filtered_string'] = [
  '#markup' => '<em>This is filtered using the admin tag list</em>',
];
$output['filtered_string'] = [
  '#markup' => '<video><source src="v.webm" type="video/webm"></video>',
  '#allowed_tags' => [
    'video',
    'source',
  ],
];
$output['escaped_string'] = [
  '#plain_text' => '<em>This is escaped</em>',
];
```

JavaScript and CSS assets are specified in the render array using the #attached property (see Attaching libraries in render arrays).

From https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/group/theme_render/10.0.x

And

```
$variables['content']['field_image']['#suffix'] = "this is a suffix to the image";
```

Or

```
$variables['content']['custom_field']= ['#type'=>'markup', '#markup'=>'Hello World'];
```

You can make up your own properties e.g. see the #selwyn_id below for use elsewhere.

```
$citation_nid = 25;
$form['actions']['accept'] = [
  '#type' => 'submit',
  '#value' => $this->t("Accept Citation $citation_nid"),
  '#citation_nid' => $citation_nid,
  '#voting_action' => 'Accept',
  '#name' => "accept_citation_$citation_nid",
  '#selwyn_id' => ['edit-accept-' . $citation_nid],
  '#attributes' => [
    'class' => [
      'hilited-button',
      'blue-button',
    ],
    'id' => ['edit-accept-' . $citation_nid],
  ],
];
```

Image

```
$image = [
  '#theme'=>'image',
  '#uri' => 'public://photo.jpg',
  '#alt' => 'hello'
];
```

Simple Text

```
$text_array = [
  '#markup' => $this->t('Hello world!'),
];
```

Text with variable substitution (Placeholders)

```
$render_array = [
  '#type' => 'markup',
  '#markup' => $this->t('You are viewing @title.  Unfortunately there is no image defined for delta: @delta.', ['@title' => $node->getTitle(), '@delta' =>$delta)],
  ];
```

And from the Render API Overview at https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/group/theme_render/10.0.x :

Placeholders in render arrays

Render arrays have a placeholder mechanism, which can be used to add data into the render array late in the rendering process. This works in a similar manner to \Drupal\Component\Render\FormattableMarkup::placeholderFormat(), with the text that ends up in the #markup property of the element at the end of the rendering process getting substitutions from placeholders that are stored in the 'placeholders' element of the #attached property.

For example, after the rest of the rendering process was done, if your render array contained:

```
$build['my_element'] = [
  '#markup' => 'Something about @foo',
  '#attached' => [
    'placeholders' => [
      '@foo' => ['#markup' => 'replacement'],
    ],
];
```

then #markup would end up containing 'Something about replacement'.

Note that each placeholder value *must* itself be a render array. It will be rendered, and any cache tags generated during rendering will be added to the cache tags for the markup.

Wrap an element with a div with a class

```
$ra['list'][$customer]['name'] = [
  '#prefix' => '<div class="customer-name">',
  '#suffix' => '</div>',
  '#type' => 'markup',
  '#markup' => $customer['name'],
];
```

Prefix and suffix

```
$ra['#prefix'] = '<div id="option-landing-block">';
$ra['#suffix'] = '</div>';
```

Date

To create a date object and return the year in a render array, use this code. Here is the code from a block build method:

```
public function build() {
  $date = new \DateTime();
  return [
    '#markup' => t('Copyright @year&copy; My Company', [
      '@year' => $date->format('Y'),
    ]), ];
}
```

This uses Drupal\Core\Datetime\DrupalDateTime which is just a wrapper for \DateTime.

Image

Load an image and display it with the alt text

```
public function displayProductImage(NodeInterface $node, $delta) {
  if (isset($node->field_product_image[$delta])) {
    $imageData = $node->field_product_image[$delta]->getValue();
    $file = File::load($imageData['target_id']);
    $render_array['image_data'] = array(
      '#theme' => 'image_style',
      '#uri' => $file->getFileUri(),
      '#style_name' => 'product_large',
      '#alt' => $imageData['alt'],
    );
  }
```

Several Url’s.

This queries for some nodes, generate a list of url’s and returns them as a render array. The '#list_type' => 'ol' (or ordered list)

```
use Drupal\Core\Url;



public function build(){
  $result = $this->nodeStorage->getQuery()
    ->accessCheck(TRUE)
    ->condition('type', 'water_action')
    ->condition('status', '1')
    ->range(0, $this->configuration['block_count'])
    ->sort('title', 'ASC')
    ->execute();

  if ($result) {
    //Only display block if there are items to show.
    $items = $this->nodeStorage->loadMultiple($result);

    $build['list'] = [
      '#theme' => 'item_list',
      '#items' => [],
    ];
foreach ($items as $item) {
  $translatedItem = $this->entityRepository->getTranslationFromContext($item);
  $nid = $item->id();
  $url = Url::fromUri("internal:/node/$nid");

  $build['list']['#items'][$item->id()] = [
    '#title' => $translatedItem->label(),
    '#type' => 'link',
    '#url' => $url,
  ];
}
```

Two paragraphs

```
$rArray = [
  'first_para' => [
    '#type' => 'markup',
    '#markup' => '...para 1 here....<br>',
  ],
  'second_para' => [
    '#type' => 'markup',
    '#markup' => '...para 2 here....<br>',

  ],
];
return $rArray;
```

A button that opens a modal dialog

```
use Drupal\Core\Url;

public function build() {
  $link_url = Url::fromRoute('custom_modal.modal');
  $link_url->setOptions([
    'attributes' => [
      'class' => ['use-ajax', 'button', 'button--small'],
      'data-dialog-type' => 'modal',
      'data-dialog-options' => Json::encode(['width' => 400]),
    ]
  ]);

  return [
    '#type' => 'markup',
    '#markup' => Link::fromTextAndUrl(t('Open modal'), $link_url)->toString(),
    '#attached' => ['library' => ['core/drupal.dialog.ajax']]
  ];
```

A link

Here is a simple link

use Drupal\Core\Url;

```
$form['noaccount'] = [
  '#type' => 'link',
  '#title' => $this->t('Continue without account'),
  '#url' => Url::fromRoute('<front>'),
];
```

Other possible urls:

```
'#url' => Url::fromUri('internal:/dashboard'),
'#url' => Url::fromUri('internal:/node/360'),
'#url' => Url::fromUri('mailto:' . $value),
```

A link with a class

Here we add the #attributes to wrap the link in the classes: button, button-action, button--primary, and button--small:

```
$form['button'] = [
  '#type' => 'link',
  '#url' => Url::fromUri('internal:/dashboard'),
  '#title' => $this->t('Go to My Training'),
  '#attributes' => ['class' => ['button', 'button-action', 'button--primary', 'button--small']],
],
```

A link and its TWIG template

Here is a link with the details of what you expect to see in the TWIG template:

```
use Drupal\Core\Url;

$back_home_link = [
  '#type' => 'link',
  '#title' => $this->t('Continue without account'),
  '#url' => Url::fromRoute('<front>'),
];

$variables['back_home_link'] = $back_home_link
```

and in the template you would expect to see something like:

```
{{ content.back_home_link }}
```

A link with parameters and a template file

This path takes a 4 parameters. Here is its path as defined in the routing.yml file:

```
team_abc.correctional_voting:
  path: '/team/abc/admin//program/{program}/expectation/{expectation}/correlation/{correlation}/{action}/{type}'
  defaults:
    _controller: '\Drupal\team_abc\Controller\CorrelationVotingController::content'
    _title: 'Correctional Voting'
  requirements:
    _permission: 'manage voting process'
  options:
    parameters:
      program:
        type: entity:node
      expectation:
        type: entity:node
      correlation:
        type: entity:node
    no_cache: 'TRUE'
```

Note the options in the routing.yml file which automatically convert the node ids to actual entities (Drupal loads the nodes internally) and passes those to the controller.

Then in the controller, we build a URL, specifying the parameters:

```
$url = Url::fromRoute('team_abc.correctional_voting', [
  'program' => $program->id(),
  'expectation' => $next_breakout_path_item['expectation_nid'],
  'correlation' => $next_breakout_path_item['correlation_nid'],
  'action' => 'vote',
  'type' => 'narrative'
]);
$next_breakout = [
  '#type' => 'link',
  '#title' => t('Next Breakout'),
  '#url' => $url,
];

// ...

$next_links[] = $next_breakout;
```

Then we wrap up all the variables and send them to buildDetails

```
  $content = [
    // ...
    'program' => $program_info,
    'previous_links' => $previous_links,
    'next_links' => $next_links,
    'expectation_cfitem_text' => strip_tags($expectation_cfitem_text),
  ];
  return $this->buildDetails($content, $breadcrumbs, $management_links, $correlation_info, $citations);
}
```

Which wraps the content in an array for rendering in twig. Note below that the `#theme` property which identifies the template filename. The #theme: `team_abc__correctional_voting` translates to the twig template file: `team-abc--correctional-voting.html.twig` where the underscores become dashes.

```
public function buildDetails(array $content, array $breadcrumbs, array $management_links, array $correlation_info, array $citations): array {
  return [
    '#theme' => 'team_abc__correctional_voting',
    '#content' => $content,
    '#breadcrumbs' => $breadcrumbs,
    '#management_links' => $management_links,
    '#correlation' => $correlation_info,
    '#citations' => $citations,
  ];
}
```

Then in team-abc–correctional-voting.html.twig the `next` links are rendered – see ``

```
<div class="cell small-12 medium-6">
  {% if content.next_links %}
    <ul class="no-bullet nav-links prev">
      {% for next_link in content.next_links %}
        Move mouse below for Next invisible links<li>{{ next_link }}</li>
      {% endfor %}
    </ul>
  {% endif %}
</div>
```

This example may be a little confusing as it loops through an array of links.

Simple unordered list

From: https://drupal.stackexchange.com/questions/214928/create-unordered-list-in-render-array

```
$content = [
  '#theme' => 'item_list',
  '#list_type' => 'ul',
  '#title' => 'My List',
  '#items' => ['item 1', 'item 2'],
  '#attributes' => ['class' => 'mylist'],
  '#wrapper_attributes' => ['class' => 'container'],
];
```

Unordered list of links for a menu

Here a list of links is created in a controller:

```
$content['tabs'] = [
    '#theme' => 'item_list',
    '#list_type' => 'ul',
    '#items' => [
    [
        '#type' => 'link',
        '#title' => $this->t('My Plumbing Training'),
        '#url' => Url::fromRoute('abc_academy.dashboard_tab', ['tab' => 'online'])
    ],
    [
        '#type' => 'link',
        '#title' => $this->t('My Enrollment Training'),
        '#url' => Url::fromRoute('abc_academy.dashboard_tab', ['tab' => 'enrollment'])
    ],
    [
        '#type' => 'link',
        '#title' => $this->t('My Transcript'),
        '#url' => Url::fromRoute('abc_academy.dashboard_tab', ['tab' => 'transcript'])
    ],
    [
        '#type' => 'link',
        '#title' => $this->t('My Certificates'),/          '#url' => Url::fromRoute('abc_academy.dashboard_tab', ['tab' => 'transcript'])
        '#url' => Url::fromUri('internal:/dashboard/certificates'),
    ],
    ]
];

return $content;
```

Nested Unordered List

```
$sidebar = [
  '#title' => 'My List',
  '#theme' => 'item_list',
  '#list_type' => 'ul',
  '#attributes' => ['class' => 'mylist'],
  '#wrapper_attributes' => ['class' => 'container'],
  '#items' => [
    [
      '#type' => 'link',
      '#title' => t('My Online Training'),
      '#url' => Url::fromUri('internal:/node/1'),
    ],
    [
      '#type' => 'link',
      '#title' => t('My Instructor-led Training'),
      '#url' => Url::fromUri('internal:/node/2'),
    ],
    ['#markup' => '<ul><li>item1</li><li>item2</li></ul>',],
    ['#markup' => '<ul><li>item1</li><li>item2</li><li>item3</li></ul>',],
  ],
];
```

Select (dropdown)

To build a select element, fill an array with some keys and text labels. The text labels will appear in the dropdown.

To set the default value i.e. the value that appears in the dropdown when it is first displayed, specify the key. For example: If the contents of

the dropdown are an array like ['aaa', 'vvv', 'zzz'] then you can specify $default=2 to display zzz as the default.

In the example below, the default is set to /node/364 and the dropdown will display Above ground pool 1, Above ground pool 2 etc.

```
// Select element.
$options = [
  '/node/360' =>'Above ground pool 1',
  '/node/362' =>'Above ground pool 2',
  '/node/364' =>'Underground pool',
  '/node/359' =>'Patio pool',
  ];

// Set the default value - it must be the key.
$default = '/node/364';
$form['select'] = [
  '#type' => 'select',
  '#title' => $this->t('Select video'),
  '#description' => 'Test Description',
  '#default_value' => $default,
  '#options' => $options,
];
```

Select (dropdown) Ajaxified

Often sites need select elements that do some action e.g. redirect when the user makes a selection in the dropdown. Here is one example such a select element. I populate the $options with the result of a database query. When the user changes the selection in the dropdown, it calls the callback videoSelectChange(). The callback redirects to the URL in question using the $command = new RedirectComand();

```
$form['select] = [
    '#type' => 'select',
//      '#title' => $this->t('Select video'),
    '#description' => 'Test Description',
    '#default_value' => $default,
    '#options' => $options,
    '#ajax' => [
    'callback' => [$this, 'videoSelectChange'],
    'event' => 'change',
    'wrapper' => $ajax_wrapper,
    ],
];



/**
 * Callback function for changes to the select elements.
 *
 */
public function videoSelectChange(array $form, FormStateInterface $form_state) {
  $values = $form_state->getValues();
  $elem = $form_state->getTriggeringElement();
  $response = new AjaxResponse();
  $url = Url::fromUri('internal:' . $values[$elem["#name"]]);
  $command = new RedirectCommand($url->toString());
  $response->addCommand($command);
  return $response;
}
```

Limit allowed tags in markup

Here we allow <i> (italics) tags in the login menu item, and <i> and <sup> (superscript) tags in the logout menu item

```
/**
 * Implements hook_preprocess_menu().
 */
function postal_theme_preprocess_menu(&$vars, $hook) {
  if ($hook == 'menu__account') {
    $items = $vars['items'];
    foreach ($items as $key => $item) {
      if ($key == 'user.page') {
        $vars['items'][$key]['title'] = [
          '#markup' => 'Log the <i>flock</i><sup>TM</sup> in!',
          '#allowed_tags' => ['i'],
        ];
      }
      if ($key == 'user.logout') {
        $vars['items'][$key]['title'] = [
          '#markup' => 'Log the <i>flock</i> <sup>TM</sup>out!',
          '#allowed_tags' => ['i', 'sup'],
        ];
      }
    }
  }
}
```

Disable an element

In this example the `accept` button is disabled when `$my_current_vote` is `accepted`.

```
  $form['accept'] = [
    '#type' => 'submit',
    '#value' => $value,
    '#name' => "accept_$feedback_error_nid",
    '#voting_action' => 'accepted',
    '#prefix' => '<div class="srp-vote-form">',
    '#id' => 'edit-accept-' . $feedback_error_nid,
    '#attributes' => [
      'class' => [
        'hilited-button',
        'blue-button',
        'accept-button',
      ],
    ],
  ];

  if (strtolower($my_current_vote) == 'accepted') {
    $form['accept']['#attributes']['class'][] = 'selected';
    $form['reject']['#disabled'] = TRUE;
  }
```

Resources

- Render API overview for Drupal 10
- Render Arrays from Drupal.org updated August 2022

Back to top

Page last modified: Jul 23 2023.

Edit this page on GitHub