Hooks

TABLE OF CONTENTS

Overview

Drupal hooks allow modules to alter and extend the behavior of Drupal core, or another module. They provide a way that code components in Drupal can communicate with one another. Using hooks, a module developer can change how core, or other modules work without changing the existing code. As a Drupal developer, understanding how to implement and invoke hooks is essential. (More at https://drupalize.me/tutorial/what-are-hooks?p=2766)

According to ChapGPT: Hooks are a key aspect of Drupal's module system, and allow developers to interact with the core functionality of Drupal 9 or Drupal 10. They can be used to alter or extend the behavior of Drupal's core features, such as adding custom validation to a form, changing the way content is displayed, or adding new actions to the administrative interface. Hooks provide a powerful and flexible way to customize Drupal to meet the needs of a specific project or site, without having to modify the core code. They are essential for developers who want to build custom modules or themes, and are a fundamental part of the Drupal development process.

Modify the login form

Here is code from hook_examples.module that modifies the user login form by adding a button. It passes the username and password that were entered to the mythical third party login endpoint.

```php
/**
 * Implements hook_form_FORM_ID_alter().
 */
function hook_examples_form_user_login_form_alter(&$form, \Drupal\Core\Form\FormStateInterface $form_state) {
  // Add a custom submit button to the login form.
  $form['third_party_login'] = [
    '#type' => 'submit',
    '#value' => t('Login with Third-Party Provider'),
    '#submit' => ['hook_examples_user_login_form_submit'], // Call the custom submit function.
  ];
}

/**
 * Custom submit function for the login form.
 */
function hook_examples_user_login_form_submit(array &$form, \Drupal\Core\Form\FormStateInterface $form_state) {
  // Get the username and password from the form state.
  $username = $form_state->getValue('name');
  $password = $form_state->getValue('pass');

  // Build the URL for the third-party login page, including the username and password as query parameters.
  $login_url = 'https://thirdpartyprovider.com/login?username=' . urlencode($username) . '&password=' . urlencode($password);

  // Redirect the user to the third-party login page.
  $form_state->setRedirect($login_url);
}
```

In the above code, the **hook_examples_form_user_login_form_alter()** function implements the **hook_form_FORM_ID_alter()** hook, where **FORM_ID** is the ID of the form being altered, in this case **user_login_form**. The function modifies the login form by adding a custom submit button, with a submit handler function of **hook_examples_user_login_form_submit()**.

When the button is clicked, the **hook_examples_user_login_form_submit()** function gets the username and password from the form state , builds the URL for the third-party login page, including the username and password as query parameters. Finally, the user is redirected to this URL using the **$form_state->setRedirect()** method.

Modify the node edit form

In this example, the save button is changed from saying "save" to "update event"

```php
use Drupal\Core\Form\FormStateInterface;
use Drupal\node\Entity\Node;

/**
 * Implements hook_form_alter().
 */
function hook_examples_form_alter(array &$form, FormStateInterface $form_state, $form_id) {
  if ($form_id === 'node_event_edit_form') {
    $node = $form_state->getFormObject()->getEntity();
    if ($node instanceof Node && $node->bundle() === 'event') {
      $form['actions']['submit']['#value'] = t('Update Event');
    }
  }
}
```

This code uses the **hook_form_alter** hook to alter the node edit form and modify the value of the submit button for nodes of type **event**. The **$form_id** argument is used to check if the form being altered is the node edit form for nodes of type **event**, and if it is, the submit button's value is changed to "Update Event". A redundant check is added to ensure the node is of type "event" for clarity.

Modify fields in a node

This example does all sorts of interesting things to the node as it is about to be saved.

It grabs some dates, fills out some fields if the user is anonymous, does some date calculations, changes the title of the node, looks up if to see if there is a connected node and grabs some info from it and updates a date field. Finally it invalidates some cache tags. Phew!

```
/**
 * Implements hook_ENTITY_TYPE_presave().
 */
function ogg_mods_node_presave(NodeInterface $node) {
  $type = $node->getType();
  if ($type == 'catalog_notice') {
    $end_date = NULL != $node->get('field_cn_start_end_dates')->end_value ? $node->get('field_cn_start_end_dates')->end_value : 'n/a';
    $govt_body = NULL != $node->field_cn_governmental_body->value ? $node->field_cn_governmental_body->value : 'Unnamed Government Body';
    $start_date_val = $node->get('field_cn_start_date')->value;

    $accountProxy = \Drupal::currentUser();
    $account = $accountProxy->getAccount();
    // Anonymous users automatically fill out the end_date.
    if (!$account->hasPermission('administer catalog notice')) {
      $days = (int) $node->get('field_cn_suspension_length')->value - 1;

      $end_date = DrupalDateTime::createFromFormat('Y-m-d', $start_date_val);
      $end_date->modify("+$days days");
      $end_date = $end_date->format("Y-m-d");
      $node->set('field_cn_end_date', $end_date);
    }

    // Always reset the title.
    $title = substr($govt_body, 0, 200) . " - $start_date_val";
    $node->setTitle($title);

    /*
     *  Fill in Initial start and end dates if this is an extension of
     * a previously submitted notice.
     */
    $extension = $node->get('field_cn_extension')->value;
    if ($extension) {
      $previous_notice_nid = $node->get('field_cn_original_notice')->target_id;
      $previous_notice = Node::load($previous_notice_nid);
      if ($previous_notice) {
        $initial_start = $previous_notice->get('field_cn_start_date')->value;
        $initial_end = $previous_notice->get('field_cn_end_date')->value;
        $node->set('field_cn_initial_start_date', $initial_start);
        $node->set('field_cn_initial_end_date', $initial_end);
      }
    }
  }
  $tags[]= 'ogg:node:' . $node->getType();
  $tags[]= 'ogg:node:' . $node->id();
  ogg_mods_invalidate_node($node);
  Cache::invalidateTags($tags);
}

/**
 * Invalidate cache associated with various nodes.
 *
 * @param \Drupal\node\NodeInterface $node
 *
 * @throws \Drupal\Component\Plugin\Exception\InvalidPluginDefinitionException
 * @throws \Drupal\Component\Plugin\Exception\PluginNotFoundException
```

```
 */
function ogg_mods_invalidate_node(NodeInterface $node) {
  $tags = [];
  $node_type = $node->getType();
  $tags[] = 'ogg:node:' . $node->id();
  switch ($node_type) {
    case 'news':
      $tags[] = 'ogg:node:home';
      $tags[] = 'ogg:node:landing_page_news';
      $tags[] = 'ogg:node:news';
      $tags[] = 'ogg:views:home_recent_news';
      break;
    default:
      break;
  }
  if (!empty($tags)) {
    Cache::invalidateTags($tags);
  }
}
```

You can read more about using hook_entity_presave at https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Entity%21entity.api.php/function/hook_entity_presave/10 as well as hook_ENTITY_TYPE_presave at https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Entity%21entity.api.php/function/hook_ENTITY_TYPE_presave/10

hook_update

Almost every project that runs for a while will require some hook_updates. This is the facility used to do automated changes to your sites. Here is an example that updates a menu item with a title 'Support'.

This code comes from a .install file:

```
function partridge_update_8002() {

  $mids = \Drupal::entityQuery('menu_link_content')
    ->condition('menu_name', 'part-wide-utility')
    ->execute();

  foreach($mids as $mid) {
    $menu_link = \Drupal::entityTypeManager()->getStorage('menu_link_content')->load($mid);

    $title = $menu_link->getTitle();
    if ($title === 'Support') {
      $menu_link->set('weight',2);
      $menu_link->set('expanded', TRUE);
      $menu_link->set('link', 'https://www.google.com');
      $menu_link->save();

    }
  }
}
```

Theme hooks

Here is an excerpt from the Theme System Overview at https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/group/themeable/10:

Preprocessing for Template Files

Several functions are called before the template file is invoked to modify the variables that are passed to the template. These make up the

"preprocessing" phase, and are executed (if they exist), in the following order (note that in the following list, HOOK indicates the hook being called or a less specific hook. For example, if '#theme'=>'node__article' is called, hook is node__article and node.

MODULE indicates a module name, THEME indicates a theme name, and ENGINE indicates a theme engine name). Modules, themes, and theme engines can provide these functions to modify how the data is preprocessed, before it is passed to the theme template:

- **template_preprocess(&$variables, $hook)**: Creates a default set of variables for all theme hooks with template implementations. Provided by Drupal Core.
- **template_preprocess_HOOK(&$variables)**: Should be implemented by the module that registers the theme hook, to set up default variables.
- **MODULE_preprocess(&$variables, $hook)**: hook_preprocess() is invoked on all implementing modules.
- **MODULE_preprocess_HOOK(&$variables)**: hook_preprocess_HOOK() is invoked on all implementing modules, so that modules that didn't define the theme hook can alter the variables.
- **ENGINE_engine_preprocess(&$variables, $hook)**: Allows the theme engine to set necessary variables for all theme hooks with template implementations.
- **ENGINE_engine_preprocess_HOOK(&$variables)**: Allows the theme engine to set necessary variables for the particular theme hook.
- **THEME_preprocess(&$variables, $hook)**: Allows the theme to set necessary variables for all theme hooks with template implementations.
- **THEME_preprocess_HOOK(&$variables)**: Allows the theme to set necessary variables specific to the particular theme hook.

Hook_preprocess

Generall, .theme files will include the following to create or alter variables for :

- **hook_preprocess_html()** the html template
- **hook_preprocess_page()** the page template
- **hook_preprocess_node()** the node template. Note hook_node_type_preprocess_node() also works where you can specify the node type e.g. wc_product_preprocess_node() which expects a content type of wc_product.

hook_preprocess_node example 1

To add a custom variable to be displayed in your template, add a function in your .theme file like the one listed below. This example also adds a #suffix to the field_image which renders that string after the field_image is rendered.

```
function mytheme_preprocess_node(&$variables) {
  $variables['custom_variable'] = "Bananas are yellow";
  $variables['content']['field_image']['#suffix'] = "this suffix on the image";
  kint($variables);
}
```

The render array you want to change will be in the content variable which shows up in the kint output as `$variables['content']`

Usually fields such as field_image will be automatically rendered by the node template (unless you've tweaked it to display in some other template.)

In your node's Twig template you would specify `` to have it display on every node.

hook_preprocess_node example 2

This code is used to make a date range like `3/30/2023 -- 3/31/2023` appear as `Mar 30-31, 2023`

The date values are stored in the field_date which is a date range field. This code is from the .theme file. Here we retrieve the starting and ending date values:

```
$from = $variables["node"]->get('field_date')->getValue()[0]['value'];
$to = $variables["node"]->get('field_date')->getValue()[0]['end_value'];
```

Here is the hook_preprocess_node()

Notice that we are creating a Twig variable called “scrunch_date” which we want to display.

```
use Drupal\Core\Datetime\DrupalDateTime;



/**
 * Implements hook_preprocess_node
 *
 * @param $variables
 */
function veryst_preprocess_node(&$variables) {
  if (!empty($variables['content']['field_date'])) {
    $date = $variables['content']['field_date'];

    $from = new DrupalDateTime($variables["node"]->get('field_date')->getValue()[0]['value']);
    $date_array = explode("-", $from);
    $from_day = substr($date_array[2], 0, 2);
    $from_month = $date_array[1];

    $to = new DrupalDateTime($variables["node"]->get('field_date')->getValue()[0]['end_value']);
    $date_array = explode("-", $to);
    $to_day = substr($date_array[2], 0, 2);
    $to_month = $date_array[1];

    if ($from_month === $to_month && $from_day != $to_day) {
      $variables['scrunch_date'] = [
        '#type' => 'markup',
        '#markup' => $from->format("M j-") . $to->format("j, Y"),
      ];
    }

  }
//  kint($variables);
}
```

Now in the twig template we can output the scrunch_date we created in the template file: web/themes/mytheme/templates/node/node--seminar--teaser.html.twig

```

```

Organizing your hooks code the OOP way

from Drupal 8 How to organise your hooks code in classes (Object-oriented way)

To implement a hook example like hook_form_FORM_ID_alter for *node_article_edit_form.*

So instead of do something like:

```php
use Drupal\Core\Form\FormStateInterface;

/**
 * Implements hook_form_BASE_FORM_ID_alter().
 */
function MY_MODULE_form_node_article_edit_form_alter(&$form, FormStateInterface $form_state, $form_id) {
  // Your code here the two following lines just an examples.
  // Hide some fields.
  $form['field_SOME_FIELD_NAME']['#access'] = FALSE;
  // Attach some library ....
  $form['#attached']['library'][] = 'MY_MODULE/SOME_LIBRARY';
}
```

You can create a class called **NodeArticleEditFormHandler** inside your **src** folder like the following:

```php
<?php

namespace Drupal\MY_MODULE;


use Drupal\Core\Form\FormStateInterface;

/**
 * Class NodeArticleEditFormHandler
 *
 * @package Drupal\MY_MODULE
 */
class NodeArticleEditFormHandler {

  /**
   * Alter Form.
   *
   * @param array $form
   *   Form array.
   * @param \Drupal\Core\Form\FormStateInterface $form_state
   *    The current state of the form.
   * @param $form_id
   *    String representing the id of the form.
   */
  public function alterForm(array &$form, FormStateInterface $form_state, $form_id) {
    // Your code here the two following lines just an examples.
    // Hide some fields.
    $form['field_SOME_FIELD_NAME']['#access'] = FALSE;
    // Attach some library ....
    $form['#attached']['library'][] = 'MY_MODULE/SOME_LIBRARY';
  }

}
```

In case you need other services you can inject your dependencies by make your class implements ContainerInjectionInterface here is an example with current user service injection:

```php
<?php

namespace Drupal\MY_MODULE;


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

```
use Drupal\Core\Session\AccountProxyInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Class NodeArticleEditFormHandler
 *
 * @package Drupal\MY_MODULE
 */
class NodeArticleEditFormHandler implements ContainerInjectionInterface {

  /**
   * The current user account.
   *
   * @var \Drupal\Core\Session\AccountProxyInterface
   */
  protected $currentUser;

  /**
   * NodeArticleEditFormHandler constructor.
   *
   * @param \Drupal\Core\Session\AccountProxyInterface $current_user
   *   The current user.
   */
  public function __construct(AccountProxyInterface $current_user) {
    $this->currentUser = $current_user;
  }

  /**
   * @inheritDoc
   */
  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('current_user')
    );
  }

  /**
   * Alter Form.
   *
   * @param array $form
   *   Form array.
   * @param \Drupal\Core\Form\FormStateInterface $form_state
   *    The current state of the form.
   * @param $form_id
   *    String representing the id of the form.
   */
  public function alterForm(array &$form, FormStateInterface $form_state, $form_id) {
    // Example to get current user.
    $currentUser = $this->currentUser;
    // Your code here the two following lines just an examples.
    // Hide some fields.
    $form['field_SOME_FIELD_NAME']['#access'] = FALSE;
    // Attach some library ....
    $form['#attached']['library'][] = 'MY_MODULE/SOME_LIBRARY';
  }

}
```

And after that change your hook into:

```php
use Drupal\MY_MODULE\NodeArticleEditFormHandler;

/**
 * Implements hook_form_BASE_FORM_ID_alter().
 */
function MY_MODULE_form_node_article_edit_form_alter(&$form, FormStateInterface $form_state, $form_id) {
  return \Drupal::service('class_resolver')
    ->getInstanceFromDefinition(NodeArticleEditFormHandler::class)
    ->alterForm($form, $form_state, $form_id);
}
```

We are done! Now your .module file is more clean, readable and maintainable with less code. You can do that with every hook for instance **EntityHandler** like:

```php
use Drupal\MY_MODULE\EntityHandler;

/**
 * Implements hook_entity_presave().
 */
function MY_MODULE_entity_presave(EntityInterface $entity) {
  return \Drupal::service('class_resolver')
    ->getInstanceFromDefinition(EntityHandler::class)
    ->entityPresave($entity);
}
```

And so on! You can see another example in Drupal core from the content moderation form

Reference

Entity hooks

Here is an excerpt from the Drupal API at https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Entity%21entity.api.php/group/entity_crud/10:

CREATE OPERATIONS

To create an entity:

$entity = $storage->**create**();

// Add code here to set properties on the entity. // Until you call save(), the entity is just in memory.

$entity->**save**();

There is also a shortcut method on entity classes, which creates an entity with an array of provided property values: \Drupal\Core\Entity::create().

Hooks invoked during the create operation:

- hook_ENTITY_TYPE_create()
- hook_entity_create()
- When handling content entities, if a new translation is added to the entity object:
  - hook_ENTITY_TYPE_translation_create()
  - hook_entity_translation_create()

See Save operations below for the save portion of the operation.

READ/LOAD OPERATIONS

To load (read) a single entity:

$entity = $storage->**load**($id);

To load multiple entities: $entities = $storage->**loadMultiple**($ids);

Since load() calls loadMultiple(), these are really the same operation. Here is the order of hooks and other operations that take place during entity loading:

- Entity is loaded from storage.
- postLoad() is called on the entity class, passing in all of the loaded entities.
- hook_entity_load()
- hook_ENTITY_TYPE_load()

When an entity is loaded, normally the default entity revision is loaded. It is also possible to load a different revision, for entities that support revisions, with this code:

$entity = $storage->**loadRevision**($revision_id);

This involves the same hooks and operations as regular entity loading.

The "latest revision" of an entity is the most recently created one, regardless of it being default or pending. If the entity is translatable, revision translations are not taken into account either. In other words, any time a new revision is created, that becomes the latest revision for the entity overall, regardless of the affected translations. To load the latest revision of an entity:

$revision_id = $storage->**getLatestRevisionId**($entity_id);

$entity = $storage->**loadRevision**($revision_id);

As usual, if the entity is translatable, this code instantiates into $entity the default translation of the revision, even if the latest revision contains only changes to a different translation:

$is_default = $entity->**isDefaultTranslation**(); // returns TRUE

The "latest translation-affected revision" is the most recently created one that affects the specified translation. For example, when a new revision introducing some changes to an English translation is saved, that becomes the new "latest revision". However, if an existing Italian translation was not affected by those changes, then the "latest translation-affected revision" for Italian remains what it was. To load the Italian translation at its latest translation-affected revision:

$revision_id = $storage->**getLatestTranslationAffectedRevisionId**($entity_id, 'it'); $it_translation = $storage ->**loadRevision**($revision_id) ->**getTranslation**('it');

SAVE OPERATIONS

To update an existing entity, you will need to load it, change properties, and then save; as described above, when creating a new entity, you will also need to save it. Here is the order of hooks and other events that happen during an entity save:

- preSave() is called on the entity object, and field objects.
- hook_ENTITY_TYPE_presave()
- hook_entity_presave()
- Entity is saved to storage.
- For updates on content entities, if there is a translation added that was not previously present:
  - hook_ENTITY_TYPE_translation_insert()
  - hook_entity_translation_insert()
- For updates on content entities, if there was a translation removed:
  - hook_ENTITY_TYPE_translation_delete()
  - hook_entity_translation_delete()
- postSave() is called on the entity object.
- hook_ENTITY_TYPE_insert() (new) or hook_ENTITY_TYPE_update() (update)
- hook_entity_insert() (new) or hook_entity_update() (update)

Some specific entity types invoke hooks during preSave() or postSave() operations. Examples:

- **Field configuration preSave()**: hook_field_storage_config_update_forbid()

- **Node postSave()**: hook_node_access_records() and hook_node_access_records_alter()
- Config entities that are acting as entity bundles in postSave(): hook_entity_bundle_create()
- **Comment**: hook_comment_publish() and hook_comment_unpublish() as appropriate.

Note that all translations available for the entity are stored during a save operation. When saving a new revision, a copy of every translation is stored, regardless of it being affected by the revision.

EDITING OPERATIONS

When an entity's add/edit form is used to add or edit an entity, there are several hooks that are invoked:

- hook_entity_prepare_form()
- hook_ENTITY_TYPE_prepare_form()
- hook_entity_form_display_alter() (for content entities only)

DELETE OPERATIONS

To delete one or more entities, load them and then delete them:

$entities = $storage->**loadMultiple**($ids);

$storage->**delete**($entities);

During the delete operation, the following hooks and other events happen:

- preDelete() is called on the entity class.
- hook_ENTITY_TYPE_predelete()
- hook_entity_predelete()
- Entity and field information is removed from storage.
- postDelete() is called on the entity class.
- hook_ENTITY_TYPE_delete()
- hook_entity_delete()

Some specific entity types invoke hooks during the delete process. Examples:

- **Entity bundle postDelete()**: hook_entity_bundle_delete()

Individual revisions of an entity can also be deleted:

$storage->**deleteRevision**($revision_id);

This operation invokes the following operations and hooks:

- Revision is loaded (see Read/Load operations above).
- Revision and field information is removed from the database.
- hook_ENTITY_TYPE_revision_delete()
- hook_entity_revision_delete()

VIEW/RENDER OPERATIONS

To make a render array for a loaded entity:

// You can omit the language ID if the default language is being used.

$build = $view_builder

->**view**($entity, 'view_mode_name', $language->**getId**());

You can also use the viewMultiple() method to view multiple entities.

Hooks invoked during the operation of building a render array:

- hook_entity_view_mode_alter()
- hook_ENTITY_TYPE_build_defaults_alter()
- hook_entity_build_defaults_alter()

View builders for some types override these hooks, notably:

- The Tour view builder does not invoke any hooks.
- The Block view builder invokes hook_block_view_alter() and hook_block_view_BASE_BLOCK_ID_alter(). Note that in other view builders, the view alter hooks are run later in the process.

During the rendering operation, the default entity viewer runs the following hooks and operations in the pre-render step:

- hook_entity_view_display_alter()
- hook_entity_prepare_view()
- Entity fields are loaded, and render arrays are built for them using their formatters.
- hook_entity_display_build_alter()
- hook_ENTITY_TYPE_view()
- hook_entity_view()
- hook_ENTITY_TYPE_view_alter()
- hook_entity_view_alter()

Some specific builders have specific hooks:

- The Node view builder invokes hook_node_links_alter().
- The Comment view builder invokes hook_comment_links_alter().

After this point in rendering, the theme system takes over. See the Theme system and render API topic for more information.

OTHER ENTITY HOOKS

Some types of entities invoke hooks for specific operations:

- Searching nodes:
  - hook_ranking()
  - Query is executed to find matching nodes
  - Resulting node is loaded
  - Node render array is built
  - comment_node_update_index() is called (this adds "N comments" text)
  - hook_node_search_result()
- Search indexing nodes:
  - Node is loaded
  - Node render array is built
  - hook_node_update_index()

Theme hooks

Here is an excerpt from the Theme System Overview at https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/group/themeable/10:

The theme system is invoked in \Drupal\Core\Render\Renderer::doRender() by calling the \Drupal\Core\Theme\ThemeManagerInterface::render() function, which operates on the concept of "theme hooks". Theme hooks define

how a particular type of data should be rendered. They are registered by modules by implementing hook_theme(), which specifies the name of the hook, the input "variables" used to provide data and options, and other information. Modules implementing hook_theme() also need to provide a default implementation for each of their theme hooks in a Twig file, and they may also provide preprocessing functions. For example, the core Search module defines a theme hook for a search result item in search_theme():

```
return array(
  'search_result' => array(
    'variables' => array(
      'result' => NULL,
      'plugin_id' => NULL,
    ),
    'file' => 'search.pages.inc',
  ),
);
```

Given this definition, the template file with the default implementation is search-result.html.twig, which can be found in the core/modules/search/templates directory, and the variables for rendering are the search result and the plugin ID. In addition, there is a function template_preprocess_search_result(), located in file search.pages.inc, which preprocesses the information from the input variables so that it can be rendered by the Twig template; the processed variables that the Twig template receives are documented in the header of the default Twig template file.

OVERRIDING THEME HOOKS

Themes may register new theme hooks within a hook_theme() implementation, but it is more common for themes to override default implementations provided by modules than to register entirely new theme hooks. Themes can override a default implementation by creating a template file with the same name as the default implementation; for example, to override the display of search results, a theme would add a file called search-result.html.twig to its templates directory. A good starting point for doing this is normally to copy the default implementation template, and then modifying it as desired.

PREPROCESSING FOR TEMPLATE FILES

Several functions are called before the template file is invoked to modify the variables that are passed to the template. These make up the "preprocessing" phase, and are executed (if they exist), in the following order (note that in the following list, HOOK indicates the hook being called or a less specific hook. For example, if '#theme'=>'node__article' is called, hook is node__article and node. MODULE indicates a module name, THEME indicates a theme name, and ENGINE indicates a theme engine name). Modules, themes, and theme engines can provide these functions to modify how the data is preprocessed, before it is passed to the theme template:

- **template_preprocess(&$variables, $hook)**: Creates a default set of variables for all theme hooks with template implementations. Provided by Drupal Core.
- **template_preprocess_HOOK(&$variables)**: Should be implemented by the module that registers the theme hook, to set up default variables.
- **MODULE_preprocess(&$variables, $hook)**: hook_preprocess() is invoked on all implementing modules.
- **MODULE_preprocess_HOOK(&$variables)**: hook_preprocess_HOOK() is invoked on all implementing modules, so that modules that didn't define the theme hook can alter the variables.
- **ENGINE_engine_preprocess(&$variables, $hook)**: Allows the theme engine to set necessary variables for all theme hooks with template implementations.
- **ENGINE_engine_preprocess_HOOK(&$variables)**: Allows the theme engine to set necessary variables for the particular theme hook.
- **THEME_preprocess(&$variables, $hook)**: Allows the theme to set necessary variables for all theme hooks with template implementations.
- **THEME_preprocess_HOOK(&$variables)**: Allows the theme to set necessary variables specific to the particular theme hook.

THEME HOOK SUGGESTIONS

In some cases, instead of calling the base theme hook implementation (either the default provided by the module that defined the hook, or the override provided by the theme), the theme system will instead look for "suggestions" of other hook names to look for. Suggestions can be specified in several ways:

- In a render array, the '#theme' property (which gives the name of the hook to use) can be an array of theme hook names instead of a single hook name. In this case, the render system will look first for the highest-priority hook name, and if no implementation is found, look for the second, and so on. Note that the highest-priority suggestion is at the end of the array.

- In a render array, the '#theme' property can be set to the name of a hook with a '__SUGGESTION' suffix. For example, in search results theming, the hook 'item_list__search_results' is given. In this case, the render system will look for theme templates called item-list--search-results.html.twig, which would only be used for rendering item lists containing search results, and if this template is not found, it will fall back to using the base item-list.html.twig template. This type of suggestion can also be combined with providing an array of theme hook names as described above.
- A module can implement hook_theme_suggestions_HOOK(). This allows the module that defines the theme template to dynamically return an array containing specific theme hook names (presumably with '__' suffixes as defined above) to use as suggestions. For example, the Search module does this in search_theme_suggestions_search_result() to suggest search_result__PLUGIN as the theme hook for search result items, where PLUGIN is the machine name of the particular search plugin type that was used for the search (such as node_search or user_search).

For further information on overriding theme hooks see https://www.drupal.org/node/2186401

ALTERING THEME HOOK SUGGESTIONS

Modules can also alter the theme suggestions provided using the mechanisms of the previous section. There are two hooks for this: the theme-hook-specific hook_theme_suggestions_HOOK_alter() and the generic hook_theme_suggestions_alter(). These hooks get the current list of suggestions as input, and can change this array (adding suggestions and removing them).

Reference Links

- What are hooks? from Drupalize.me March 2022
- Theme system overview on api.drupal.org
- How to organize your hooks the object oriented way by Azz-eddine BERRAMOU Mar 2020

Back to top

Page last modified: May 14 2023.

Edit this page on GitHub