Customizer Objects

- Settings
- Controls
  - Core Custom Controls
- Sections
- Panels
- Custom Controls, Sections, and Panels
  - Customizer UI Standards
- Custom Setting Types

The Customize API is object-oriented. There are four main types of Customizer objects: Panels, Sections, Settings, and Controls. Settings associate UI elements (controls) with settings that are saved in the database. Sections are UI containers for controls, to improve their organization. Panels are containers for sections, allowing multiple sections to be grouped together. Each Customizer object is represented by a PHP class, and all of the objects are managed by the Customize Manager object, WP_Customize_Manager.
Graphic showing the relationship between each type of customize object To add, remove, or modify any Customizer object, and to access the Customizer Manager, use the `customize_register` hook:

```
function themeslug_customize_register( $wp_customize ) {
  // Do stuff with $wp_customize, the WP_Customize_Manager object.
}
add_action( 'customize_register', 'themeslug_customize_register' );
```

The Customizer Manager provides add_, get_, and remove_ methods for each Customizer object type; each works with an id. The get_ methods allow for direct modification of parameters specified when adding a control.

```
add_action('customize_register','my_customize_register');
function my_customize_register( $wp_customize ) {
  $wp_customize->add_panel();
  $wp_customize->get_panel();
  $wp_customize->remove_panel();
```

```
  $wp_customize->add_section();
  $wp_customize->get_section();
  $wp_customize->remove_section();

  $wp_customize->add_setting();
  $wp_customize->get_setting();
  $wp_customize->remove_setting();

  $wp_customize->add_control();
  $wp_customize->get_control();
  $wp_customize->remove_control();
}
```

Note: themes generally should not modify core sections and panels with the get methods, since themes should not modify core, theme-agnostic functionality. Plugins are encouraged to use these functions where necessary. Themes should not “reorganize” customizer sections that aren’t added by the theme.

Settings

Settings handle live-previewing, saving, and sanitization of your customizer objects. Each setting is managed by a control object. There are several parameters available when adding a new setting:

```
$wp_customize->add_setting( 'setting_id', array(
  'type' => 'theme_mod', // or 'option'
  'capability' => 'edit_theme_options',
  'theme_supports' => '', // Rarely needed.
  'default' => '',
  'transport' => 'refresh', // or postMessage
  'sanitize_callback' => '',
  'sanitize_js_callback' => '', // Basically to_json.
) );
```

Important: Do *not* use a setting ID that looks like `widget_*`, `sidebars_widgets[*]`, `nav_menu[*]`, or `nav_menu_item[*]`. These setting ID patterns are reserved for widget instances, sidebars, nav menus, and nav menu items respectively. If you need to use “widget” in your setting ID, use it as a suffix instead of a prefix, for example “`homepage_widget`”.

There are two primary types of settings: options and theme modifications. Options are stored directly in the wp_options table of the WordPress database and apply to the site regardless of the active theme. Themes should rarely if ever add settings of the option type. Theme mods, on the

other hand, are specific to a particular theme. Most theme options should be theme_mods. For example, a custom CSS plugin could register a custom theme css setting as a theme_mod, allowing each theme to have a unique set of CSS rules without losing the CSS when switching themes then switching back. customize-theme-mods-options

Theme_mod vs. option setting type example.

It is usually most important to set the default value of the setting as well as its sanitization callback, which will ensure that no unsafe data is stored in the database. Typical theme usage:

```
$wp_customize->add_setting( 'accent_color', array(
  'default' => '#f72525',
  'sanitize_callback' => 'sanitize_hex_color',
) );
```

Typical plugin usage:

```
$wp_customize->add_setting( 'myplugin_options[color]', array(
  'type' => 'option',
  'capability' => 'manage_options',
  'default' => '#ff2525',
  'sanitize_callback' => 'sanitize_hex_color',
) );
```

Note that the Customizer can handle options stored as keyed arrays for settings using the option type. This allows multiple settings to be stored in a single option that isn't a theme mod. To retrieve and use the values of your Customizer options, use `get_theme_mod()` and `get_option()` with the setting id:

```
function my_custom_css_output() {
  echo '<style type="text/css" id="custom-theme-css">' .
  get_theme_mod( 'custom_theme_css', '' ) . '</style>';
  echo '<style type="text/css" id="custom-plugin-css">' .
  get_option( 'custom_plugin_css', '' ) . '</style>';
}
add_action( 'wp_head', 'my_custom_css_output');
```

Note that the second argument for `get_theme_mod()` and `get_option()` is the default value, which should match the default you set when adding the setting.

Controls

Controls are the primary Customizer object for creating UI. Specifically, every control must be associated with a setting, and that setting will save user-entered data from the control to the database (in addition to displaying it in the live-preview and sanitizing it). Controls can be added by the Customizer Manager and provide a robust set of UI options with minimal effort:

```
$wp_customize->add_control( 'setting_id', array(
  'type' => 'date',
  'priority' => 10, // Within the section.
  'section' => 'colors', // Required, core or custom.
  'label' => __( 'Date' ),
  'description' => __( 'This is a date control with a red border.' ),
  'input_attrs' => array(
    'class' => 'my-custom-class-for-js',
    'style' => 'border: 1px solid #900',
    'placeholder' => __( 'mm/dd/yyyy' ),
  ),
  'active_callback' => 'is_front_page',
) );
```

The most important parameter when adding a control is its type — this determines what type of UI the Customizer will display. Core provides several built-in control types:

- `<input>` elements with any allowed type (see below)
- `checkbox`
- `textarea`
- `radio` (pass a keyed array of values => labels to the `choices` argument)
- `select` (pass a keyed array of values => labels to the `choices` argument)
- `dropdown-pages` (use the `allow_addition` argument to allow users to add new pages from the control)

For any input type supported by the html `input` element, simply pass the type attribute value to the type parameter when adding the control. This allows support for control types such as `text`, `hidden`, `number`, `range`, `url`, `tel`, `email`, `search`, `time`, `date`, `datetime`, and `week`, pending browser support. Controls must be added to a section before they will be displayed (and sections must contain controls to be displayed). This is done by specifying the section parameter when adding the control. Here is an example for adding a basic textarea control:

```
$wp_customize->add_control( 'custom_theme_css', array(
  'label' => __( 'Custom Theme CSS' ),
  'type' => 'textarea',
  'section' => 'custom_css',
) );
```

And here’s an example of a basic range (slider) control. Note that most browsers will not display the numeric value of this control because the range input type is designed for settings where the exact value is unimportant. If the numeric value is important, consider using the number type. The `input_attrs` parameter will map a keyed array of attributes => values to attributes on the input element, and can be used for purposes ranging from placeholder text to `data-` JavaScript-referenced data in custom scripts. For number and range controls, it allows us to set the minimum, maximum, and step values.

```
$wp_customize->add_control( 'setting_id', array(
  'type' => 'range',
  'section' => 'title_tagline',
  'label' => __( 'Range' ),
  'description' => __( 'This is the range control description.' ),
  'input_attrs' => array(
    'min' => 0,
    'max' => 10,
    'step' => 2,
  ),
) );
```

Core Custom Controls

If none of the basic control types suit your needs, you can easily create and add custom controls. Custom controls are explained more fully later in this post, but they are essentially subclasses of the base `WP_Customize_Control` object that allow any arbitrary html markup and functionality that you might need. Core features several built-in custom controls that allow developers to implement rich JavaScript-driven features with ease. A color picker control can be added as follows:

```
$wp_customize->add_control( new WP_Customize_Color_Control( $wp_customize, 'color_control',
array(
  'label' => __( 'Accent Color', 'theme_textdomain' ),
  'section' => 'media',
) ) );
```

WordPress 4.1 and 4.2 also added support for any type of multimedia content, with the Media control. The media control implements the native WordPress media manager, allowing users to select files from their library or upload new ones. By specifying the `mime_type` parameter when adding the control, you can instruct the media library show to a specific type such as images or audio:

```
$wp_customize->add_control( new WP_Customize_Media_Control( $wp_customize, 'image_control',
array(
  'label' => __( 'Featured Home Page Image', 'theme_textdomain' ),
  'section' => 'media',
  'mime_type' => 'image',
) ) );
```

```
$wp_customize->add_control( new WP_Customize_Media_Control( $wp_customize, 'audio_control',
array(
  'label' => _( 'Featured Home Page Recording', 'theme_textdomain' ),
  'section' => 'media',
  'mime_type' => 'audio',
) ) );
```

Note that settings associated with `WP_Customize_Media_Control` save the associated attachment ID, while all other media-related controls (children of `WP_Customize_Upload_Control`) save the media file URL to the setting. More information is available on Make WordPress Core. Additionally, WordPress 4.3 introduced the `WP_Customize_Cropped_Image_Control`, which provides an interface for cropping an image after selecting it. This is useful for instances where a particular aspect ratio is needed.

Sections

Sections are UI containers for Customizer controls. While you can add custom controls to the core sections, if you have more than a few options you may want to add one or more custom sections.

Use the `add_section` method of the WP_Customize_Manager object to add a new section:

```
$wp_customize->add_section( 'custom_css', array(
  'title' => __( 'Custom CSS' ),
  'description' => __( 'Add custom CSS here' ),
  'panel' => '', // Not typically needed.
  'priority' => 160,
  'capability' => 'edit_theme_options',
  'theme_supports' => '', // Rarely needed.
) );
```

You only need to include fields that you want to override the default values of. For example, the default priority (order of appearance) is typically acceptable, and most sections shouldn't require descriptive text if your options are self-explanatory. If you do want to change the location of your custom section, the priorities of the core sections are below:

Title	ID	Priority (Order)
Site Title & Tagline	title_tagline	20
Colors	colors	40
Header Image	header_image	60
Background Image	background_image	80
Menus (Panel)	nav_menus	100
Widgets (Panel)	widgets	110
Static Front Page	static_front_page	120
default		160
Additional CSS	custom_css	200

In most cases, sections can be added with only one or two parameters being specified. Here's an example for adding a section for options that pertain to a theme's footer:

```
// Add a footer/copyright information section.
$wp_customize->add_section( 'footer' , array(
  'title' => __( 'Footer', 'themename' ),
  'priority' => 105, // Before Widgets.
) );
```

Panels

The Customizer Panels API was introduced in WordPress 4.0, and allows developers to create an additional layer of hierarchy beyond controls and sections. More than simply grouping sections of controls, panels are designed to provide distinct contexts for the Customizer, such as Customizing Widgets, Menus, or perhaps in the future, editing posts. There is an important technical distinction between the section and panel objects. **Themes should not register their own panels in most cases**. Sections do *not* need to be nested under a panel, and each section should generally contain multiple controls. Controls should also be added to the Sections that core provides, such as adding color options to the colors Section. Also make sure that your options are as streamlined and efficient as possible; see the WordPress philosophy. Panels are designed as contexts for entire features such as Widgets, Menus, or Posts, not as wrappers for generic sections. If you absolutely must use Panels, you'll find that the API is nearly identical to that for Sections:

```
$wp_customize->add_panel( 'menus', array(
  'title' => __( 'Menus' ),
  'description' => $description, // Include html tags such as <p>.
  'priority' => 160, // Mixed with top-level-section hierarchy.
```

```
) );
$wp_customize->add_section( $section_id , array(
  'title' => $menu->name,
  'panel' => 'menus',
) );
```

Panels must contain at least one Section, which must contain at least one Control, to be displayed. As you can see in the above example, Sections can be added to Panels similarly to how Controls are added to Sections. However, unlike with controls, if the Panel parameter is empty when registering a Section, it will be displayed on the main, top-level Customizer context, as most sections should not be contained with a panel.

Custom Controls, Sections, and Panels

Custom Controls, Sections, and Panels can be easily created by subclassing the PHP objects associated with each Customizer object: `WP_Customize_Control`, `WP_Customize_Section`, and `WP_Customize_Panel` (this can also be done for `WP_Customize_Setting`, but custom settings are typically better implemented using custom setting types, as outlined in the next section). Here's an example for a basic custom control:

```
class WP_New_Menu_Customize_Control extends WP_Customize_Control {
  public $type = 'new_menu';
  /**
  * Render the control's content.
  */
  public function render_content() {
  ?>
    <button class="button button-primary" id="create-new-menu-submit" tabindex="0"><?php _e(
'Create Menu' ); ?></button>
  <?php
  }
}
```

By subclassing the base control class, you can override any functionality with custom functionality or use the core functionality depending on your needs. The most common function to override is `render_content()` as it allows you to create custom UI from scratch with HTML. Custom Controls should be used with caution, however, as they may introduce UI that is inconsistent with the surrounding core UI and cause confusion for users. Custom Customizer objects can be added

similarly to how default controls, sections, and panels are added:

```
$wp_customize->add_control(
  new WP_Customize_Color_Control(
    $wp_customize, // WP_Customize_Manager
    'accent_color', // Setting id
    array( // Args, including any custom ones.
      'label' => __( 'Accent Color' ),
      'section' => 'colors',
    )
  )
);
```

Parameters passed when adding controls are mapped to class variables within the control class, so you can add and use custom ones where certain parts of your custom object are different across different instances. When creating custom Controls, Sections, or Panels, it is strongly recommended to reference the core code, in order to fully understand the available functionality that can be overridden. Core also includes examples of custom objects of each type. This can be found in wp-includes/class-wp-customize-control.php, wp-includes/class-wp-customize-section.php, and wp-includes/class-wp-customize-panel.php. There is also a JavaScript API for each Customizer object type, which can be extended with custom objects; see the Customizer JavaScript API section for more details.

Customizer UI Standards

Custom customizer controls, sections, and panels should match core UI practices wherever possible. This includes relying on standards from wp-admin, such as using the `.button` and `.button-primary` classes, for example. There are also a few standards specific to the customizer (as of WordPress 4.7):

- White background colors are used *only* to indicate navigation and actionable items (such as inputs)
- The general `#eee` background color provides visual contrast against the white elements
- `1px #ddd` borders separate navigational elements from background margins and from each other
- `15px` of spacing is provided between elements where visual separation is desired
- `4px` borders are used on one side of a navigation element to show hover or focus, with a color of `#0073aa`
- Customizer text uses `color: #555d66`, with `#0073aa` for hover and focus states on navigation elements

Custom Setting Types

By default, the Customizer supports saving settings as options or theme modifications. But this behavior can be easily overwritten to manually save and preview settings outside of the wp_options table of the WordPress database, or to apply other custom handling. To get started, specify a type other than option or theme_mod when adding your setting (you can use pretty much any string):

```
$wp_customize->add_setting( $nav_menu_setting_id, array(
  'type' => 'nav_menu',
  'default' => $item_ids,
) );
```

The setting will no longer be saved or previewed when its value is changed in the associated control. Now, you can use the `customize_update_$setting->type` and `customize_preview_$setting->type` actions to implement custom saving and previewing functionality. Here is an example for saving a menu item's order property from the Menu Customizer project (the value of the setting is an ordered array of menu ids):

```
function menu_customizer_update_nav_menu( $value, $setting ) {
  $menu_id = str_replace( 'nav_menu_', '', $setting->id );
  // ...
  $i = 0;
  foreach( $value as $item_id ) { // $value is ordered array of item ids.
    menu_customizer_update_menu_item_order( $menu_id, $item_id, $i );
  $i++;
  }
}
add_action( 'customize_update_nav_menu', 'menu_customizer_update_nav_menu', 10, 2 );
```

And here is how the same plugin implements previewing for nav menu items (note that this example requires PHP 5.3 or higher):

```
function menu_customizer_preview_nav_menu( $setting ) {
  $menu_id = str_replace( 'nav_menu_', '', $setting->id );
  add_filter( 'wp_get_nav_menu_items', function( $items, $menu, $args ) use ( $menu_id,
$setting ) {
    $preview_menu_id = $menu->term_id;
    if ( $menu_id == $preview_menu_id ) {
      $new_ids = $setting->post_value();
      foreach ( $new_ids as $item_id ) {
```

```
        $item = wp_setup_nav_menu_item( $item );
        $item->menu_order = $i;
        $new_items[] = $item;
        $i++;
      }
      return $new_items;
    } else {
      return $items;
    }
  }, 10, 3 );
}
add_action( 'customize_preview_nav_menu', 'menu_customizer_preview_nav_menu', 10, 2 );
```

□□□ #2
□ □□□ □□□ 2026-08-29 15:36:32 UTC
□ □□□ □□□ 2026-08-29 15:36:32 UTC