JavaScript/Underscore.js-Rendered Custom Controls

- Registered Control Types
- Sending PHP Control Data to JavaScript
- JS/Underscore Templating
- Putting the pieces together

WordPress 4.1 also added support for rendering JavaScript-heavy and/or high-quantity controls entirely with JavaScript. This allows for more dynamic behavior, especially related to dynamically-added controls. The core Color and Media controls currently leverage this API, and all core Controls will eventually use it in the future. The PHP-based control API is not going away, but in the future most controls will likely use the new API since it provides a faster experience for users and developers. Similar APIs for JS-templated Sections and Panels were introduced WordPress 4.3; however, there remain some gaps in the ease of dynamically-creating objects in JS as of WordPress 4.7, see #30741.

Registered Control Types

In order to introduce a concept of having one template for multiple Customizer controls of the same type, we needed to introduce a way to register a type of control with the Customize Manager. Previously, custom control objects were only encountered when custom controls were added using `WP_Customize_Manager::add_control()`. But detecting added control types to render one template per type wouldn't allow new controls to be created dynamically if no other instances of that type were loaded. `WP_Customize_Manager::register_control_type() solves this:`

```
add_action( 'customize_register', 'prefix_customize_register' );
function prefix_customize_register( $wp_customize ) {
  // Define a custom control class, WP_Customize_Custom_Control.
  // Register the class so that its JS template is available in the Customizer.
  $wp_customize->register_control_type( 'WP_Customize_Custom_Control' );
}
```

All registered control types have their templates printed to the customizer by `WP_Customize_Manager::print_control_templates()`.

Sending PHP Control Data to JavaScript

While Customizer control data has always been passed to the control JS models, and this has always been able to be extended, you're much more likely to need to send data down when working with JS templates. Anything that you would want access to in `render_content()` in PHP will need to be exported to JavaScript to be accessible in your control template. `WP_Customize_Control` exports the following control class variables by default:

- `type`
- `label`
- `description`
- `active` (boolean state)

You can add additional parameters specific to your custom control by overriding `WP_Customize_Control::to_json()` in your custom control subclass. In most cases, you'll want to call the parent class' `to_json()` method also, to ensure that all core variables are exported as well. Here's an example from the core color control:

```
public function to_json() {
  parent::to_json();
  $this->json['statuses'] = $this->statuses;
  $this->json['defaultValue'] = $this->setting->default;
}
```

JS/Underscore Templating

Once you've registered your custom control class as a control type and exported any custom class variables, you can create the template that will render the control UI. You'll override `WP_Customize_Control::content_template()` (empty by default) as a replacement for `WP_Customize_Control::render_content()`. Render content is still called, so be sure to override it with an empty function in your subclass as well.

Underscore-style custom control templates are very similar to PHP. As more and more of WordPress core becomes JavaScript-driven, these templates are becoming increasingly more common. Some sample template code in core can be found in

media, revisions, the theme browser, and even in the Twenty Fifteen theme, where a JS template is used to both save the color scheme data and instantly preview color scheme changes in the Customizer. The best way to learn how these templates work is to study similar code in core and, accordingly, here is a brief example:

```
class WP_Customize_Color_Control extends WP_Customize_Control {
  public $type = 'color';
// ...
  /**
   * Render a JS template for the content of the color picker control.
   */
  public function content_template() {
    ?>
    <# var defaultValue = '';
    if ( data.defaultValue ) {
      if ( '#' !== data.defaultValue.substring( 0, 1 ) ) {
        defaultValue = '#' + data.defaultValue;
      } else {
        defaultValue = data.defaultValue;
      }
      defaultValue = ' data-default-color=' + defaultValue; // Quotes added automatically.
    } #>
    <label>
      <# if ( data.label ) { #>
        <span class="customize-control-title">{{{ data.label }}}</span>
      <# } #>
      <# if ( data.description ) { #>
        <span class="description customize-control-description">{{{ data.description
}}}</span>
      <# } #>
      <div class="customize-control-content">
        <input class="color-picker-hex" type="text" maxlength="7" placeholder="<?php
esc_attr_e( 'Hex Value' ); ?>" {{ defaultValue }} />
      </div>
    </label>
    <?php
  }
}
```

In the above template for the core custom color control, you can see that after the closing PHP tag, we have a JS template. notation is used around statements to be evaluated – in most cases, this is used for conditionals. All of the control instance data that we exported to JS is stored in the `data` object, and we can print a variable using double (escaped) or triple (unescaped) brace notation `{{ }}`. As I said before, the best way to get the hang of writing controls like this is to read through existing examples. `WP_Customize_Upload_Control` was recently updated to leverage this API as well, integrating nicely with the way the media manager is implemented, and squeezing a ton of functionality out of a minimal amount of code. If you want some really good practice, try converting some of the other core controls to use this API – and submit patches to core too, of course!

Putting the pieces together

Here's a summary of what's needed to leverage the new API in a custom customizer control subclass:

1. Make your render_content() function empty (but it does need to exist to override the default one).
2. Create a new function, content_template(), and put the old contents of render_content() there.
3. Add any custom class variables that are needed for the control to be exported to the JavaScript in the browser (the JSON data) by modifying the to_json() function (see WP_Customize_Color_Control for an example).
4. Convert the PHP from render_content() into a JS template, using around JS statements to evaluate and {{ }} around variables to print. PHP class variables are available in the data object; for example, the label can be printed with {{ data.label }}.
5. **Register the custom control class/type**. This critical step tells the Customizer to print the template for this control. This is distinct from just printing templates for all controls that were added because the ideas are that many instances of this control type could be rendered from one template, and that any registered control types would be available for dynamic control-creation in the future. Just do something like $wp_customize->register_control_type('WP_Customize_Color_Control');.

The PHP-only parts of the Customize API are still fully supported and perfectly fine to use. But given long term goals for making the customizer more flexible for doing things like switching themes in the customizer without a pageload, it is strongly encouraged to use JS/Underscore templates for all custom customizer objects where feasible.

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