The WordPress Theme Customizer: a Comprehensive Developer's Guide

November 5, 2012

This tutorial will explain in detail how to add support for the WordPress theme customizer to your WordPress theme.

The theme customizer was introduced in WordPress version 3.4. It allows for an editing environment where theme options can be tried by the administrator before being applied to the live site. In this tutorial, we will look at exactly how this feature can be added to a theme. The WordPress theme we will use for this example will be the Responsive theme version 1.8.0.1, by Emil Uzelac. This is one of the featured themes on WordPress.org at the moment and should give us a solid starting point. However, please feel free to use whatever theme you'd like as you follow along.

Please note: There are a few different ways to implement the theme customizer and save the customization settings. This tutorial will focus on the theme_mod method. If you don't know what that means, that's okay. You don't need to have any understanding the different methods to follow along with this tutorial.

1. Add the Theme Customizer page to the admin menu

Note: Step one is no longer necessary with new versions of WordPress. The customizer is automatically added to the menu even if the theme doesn't use it. Feel free to skip to step two.

First we'll open up the theme's functions.php file and add the following code. It doesn't really matter where in the functions.php file we place the code as long as it isn't inside another function.

```
1  /**
2   * Adds the Customize page to the WordPress admin area
3   */
4  function example_customizer_menu() {
5      add_theme_page( 'Customize', 'Customize', 'edit_theme_opt
```

```
}
add_action( 'admin_menu', 'example_customizer_menu' );
```

Here we create a new function title "example_customizer_menu" and attach it to the "admin_menu" action hook (If you don't understand WordPress action hooks, I would suggest reading Nathan Rice's excellent Introduction to WordPress Action Hooks post). This function has only one line, and it calls the add_theme_page() function for adding pages to the WordPress admin area. This function has four required parameters.

Function: add_theme_page($page_title, $menu_title, $capability, $menu_slug, $function);

$page_title	string	This will be the title text of the new page. However, the customizer page doesn't have a title, so I suppose this doesn't matter too much.
$menu_title	string	This will be the text of the new menu item in the "Appearance" menu.
$capability	string	This makes sure the user has the proper permissions to access the customizer (For more on WordPress user
$menu_slug	string	The unique menu slug for this page.

Please note: If you are building a theme for public release, $page_title and $menu_title should be translation friendly. For more details on internationalization, please see the I18n for WordPress page of the codex.

Now if we go back to the WordPress admin area there will be a new page titled "Customize" available in the "Appearance" menu. Clicking the Appearance > Customize link will take us to the new theme customizer page we just created. We will notice there are already four customization sections listed in the panel on the left side of the screen. "Site Title and Tagline" as well as "Static Front Page" are added to the customizer automatically. The "Colors" and "Background Image" sections are added because the theme supports the custom background feature built into the WordPress core. In the next section we will add our own settings section.

2. Add a new section to the customizer

To create our own settings section, we will place the following piece of code in the functions.php file just below the code we added earlier (once again, the placement doesn't matter a whole lot, it's just easier to add the code in order).

```
/**
 * Adds the individual sections, settings, and controls to t
 */
function example_customizer( $wp_customize ) {
    $wp_customize->add_section(
```

```
            'example_section_one',
            array(
                'title' => 'Example Settings',
                'description' => 'This is a settings section.',
                'priority' => 35,
            )
        );
    }
    add_action( 'customize_register', 'example_customizer' );
```

Please note: You will not be able to see the settings section until it contains at least one setting.

We have attached our function to the "customize_register" action hook to ensure that it runs at the proper time. The $wp_customize object is passed to our function and we can use its methods to add sections, settings and controls to the theme customizer. We'll start by adding a section using the add_section() method. This method accepts two parameters.

Method: add_section($id, $args);		
$id	string	The unique ID of this section.
$args	array	The array of arguments passed to the add_section() function.

The $args array allows us to specify a number of details about the settings section we're creating.

Array: $args		
title	string	Title of the settings section.
description	string	Optional. Description of the settings section. This is only displayed as a tooltip when the mouse hovers over the section title.
priority	integer	Optional. The priority determines the order in which the sections will be displayed. Lower numbers come first. Default: 10.
capability	string	Optional. Show or hide section based on the user's permission levels.
theme_supports	string	Optional. Show or hide the section based on whether the theme supports a particular feature.

In our example above, we have not used the optional "capability" argument or the "theme_supports" argument. However, the optional "description" and "priority" are used so you can get a feel for how they function. Go ahead and change their values to see how they work. Next we will create a setting to be placed in our newly created section.

3. Add a new setting

Now we will add a new setting to the settings section we just created. This is a two part process. First, we must register the setting. Second, we must create a control that displays the setting. We will start by creating a text setting that will be used to display the copyright information in the theme's footer.

3.1 Register a setting

Settings are registered by using the add_setting() method. We will place it just after the add_section() call and just before the closing curly brace of the example_customizer() function.

```
$wp_customize->add_setting(
    'copyright_textbox',
    array(
        'default' => 'Default copyright text',
    )
);
```

Please note: You will not be able to see this new setting until it has it's own control. Don't worry, we'll cover controls in a moment.

Just like the add_section() method, the add_setting() method accepts two parameters, an ID and an array of arguments.

Method: add_setting($id, $args)		
$id	string	The unique ID of this setting.
$args	array	The array of arguments passed to the add_setting() function.

...and here are are the arguments that can be passed in the $args array.

Array: $args		
default	string	Optional. The default value of the setting if no value has already been saved. Default: blank.
type	string	Optional. Determines how the setting will be saved. Default: theme_mod.
capability	string	Optional. Show or hide the setting based on the user's permission levels. Default: edit_theme_options.
theme_supports	string	Optional. Show or hide the section based on whether the theme supports a particular feature. Default: blank.
transport	string	Optional. Determines how the new setting values will be transfered to the live preview. Default: refresh.
sanitize_callback	string	Optional. The name of the function that will

		be called to sanitize the input data before saving it to the database. Default: blank.
sanitize_js_callback	string	Optional. The name of the function that will be called to sanitize the coming from the database on its way to the theme customizer. Default: blank.

You'll notice in the code above we're only using the "default" argument. Most of the other arguments have acceptable defaults, so we won't bother adding them to our add_setting() method call. There are a few arguments that deserve a little extra attention before we move on.

First, the "sanitize_callback" argument should be used to verify the data that has been entered before storing it in the database. For the sake of simplicity, I've left this step out for now but I'll come back to it later on in the tutorial. I believe that data sanitization is often overlooked and I want to make sure to give it the attention that it deserves.

Second, the "sanitize_js_callback" might not be what you would expect. I typically think of data sanitization as something that happens to user input before it is placed in the database. However, the "sanitize_js_callback" function is called to sanitize the data stored in the database as it is passed to the theme customizer. The only use case I could find for this function was to add "#" to the beginning of a hexadecimal value in case it was stored in the database without it.

Third, the "transport" argument is also worth mentioning. When the value of our setting is changed in the customizer, there are two ways of updating the live preview to reflect the new input. The "refresh" option refreshes the frame displaying the live preview. This happens automatically. However, if that little delay while the page refreshes is unacceptable, you can use the "postMessage" option. This passes the new value to the live preview using AJAX and removes the need for a page refresh. In order to use "postMessage" you'll need to write your own scripts to handle the AJAX data being passed to the live preview. For the sake of simplicity, we'll use the default "refresh" transport option for now and return to cover the "postMessage" transport option later on in the tutorial.

Now that that's behind us, we can finally add a control to our setting.

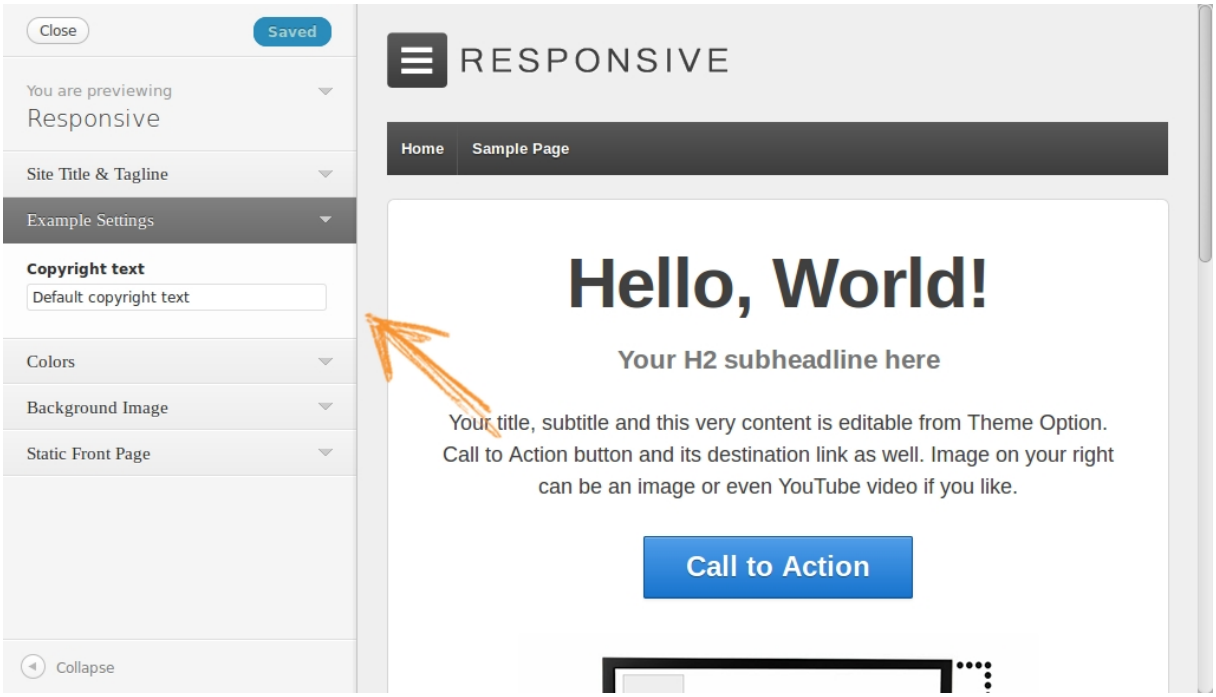
3.2 Creating a control

Controls are simply the interface used to change a setting. Each setting must have a control in order to be visible in the theme customizer. A control is added using the add_control() method. Just like our previous methods, it takes two parameters. The first is the ID of the control. This should match the ID of the setting that the control belongs to. Since our setting above had an ID of "textbox", the control must also have "textbox" as its control. Just like before, the

second parameter is an argument array. We will place our call to the add_control() method just below our add_setting() function call and just before the closing curly brace of the example_customizer() function.

```
$wp_customize->add_control(
    'copyright_textbox',
    array(
        'label' => 'Copyright text',
        'section' => 'example_section_one',
        'type' => 'text',
    )
);
```

Now we should finally be able to see the section and the setting in the theme customizer!



Just like the previous methods, the add_control() method takes two parameters, the ID and the arguments array.

Method: add_control($id, $args)

$id	string	The ID should match the ID of the setting that the control belongs with.
$args	array	The array of arguments passed to the add_control() function.

Here are the arguments that can be placed in the $args array.

Array: $args

label	string	Label text for the setting.
section	string	The ID of the section where this setting will be placed
type	string	Optional: The type of input this setting should use. Default: text.

choices	array	If this setting uses a select list or radio buttons, the "choices" argument specifies the list of options to be displayed.
priority	integer	Optional. The priority determines the order in which the settings will be displayed within the section. Lower numbers come first. Default: 10.

Most of these arguments are pretty straight forward. As you can see, we only used the first three in the example above. We will deal with the "choices" argument later on when we add select lists and radio buttons. The "priority" argument was also left off in the example because we have only one settings field. Specifying a priority will not affect the settings order until there are multiple settings. We will get to this in the next section when add other types of settings fields to the theme customizer.

4. Access the stored setting from the theme

At last! We can finally use the new setting in our theme. As mentioned earlier, we will be using this setting to control the copyright text in the footer. The current copyright notice is displayed by this line in the footer.php file of the theme:

```
1  <?php bloginfo('name'); ?>
```

We want to remove that line and replace it with the following line:

```
1  <?php echo get_theme_mod( 'copyright_textbox', 'No copyright
```

The get_theme_mod() function accepts two parameters.

Function: get_theme_mod($name, $default)		
$name	string	The name of the setting to retrieve. The name of the setting is whatever string you specified as the setting ID when you created the setting.
$default	string	Optional. The default string of text to be displayed if no setting has been saved.

You'll notice that the first parameter in the code above is the same as the ID of the setting we created earlier. This is how WordPress knows which value to grab out of the database. The second parameter is just the fallback text to display just in case the "copyright_textbox" setting hasn't been saved yet.

5. Additional control types

We've already created a textbox setting and added it to our theme, but that's just the beginning. There are a number of other standard control types provided by WordPress.

5.1 Checkbox

We will use this setting to allow the copyright information to be hidden entirely. First, we will create another setting by adding the following code just below the text control we added earlier.

```
$wp_customize->add_setting(
    'hide_copyright'
);
```

Notice that unlike the text setting we created earlier, we don't need an arguments array because we don't need a default setting value. Next we'll add the code for the checkbox control below the setting we just added.

```
$wp_customize->add_control(
    'hide_copyright',
    array(
        'type' => 'checkbox',
        'label' => 'Hide copyright text',
        'section' => 'example_section_one',
    )
);
```

The only difference between this control and the text control is we've added the "type" property to the arguments array. This lets WordPress know that we want a checkbox to be displayed rather than the default text input field. Now lets use this in the theme to hide the copyright text if the checkbox is checked. Here is the default copyright code from the footer.php file of the theme:

```
<?php esc_attr_e('©', 'responsive'); ?> <?php _e(date('Y'));
    <?php echo get_theme_mod( 'copyright_textbox', 'No copyri
</a>
```

We want to check the hide_copyright setting and only display the copyright if the checkbox is blank. The theme customizer stores the value of the checkbox as the numeral 1 if it is checked and blank if it isn't checked. So we'll add an if statement to make sure the hide_copyright checkbox is blank before displaying the copyright text. The original code above will be replaced by the following section:

```
<?php if( get_theme_mod( 'hide_copyright' ) == '') { ?>
    <?php esc_attr_e('©', 'responsive'); ?> <?php _e(date('Y'
        <?php echo get_theme_mod( 'copyright_textbox', 'No co
    </a>
<?php } // end if ?>
```

5.2 Radio buttons

To demonstrate the use of radio buttons, we will create a new customizer option to specify the placement of the logo, either left, right, or center. First let's create the setting and control by adding the following block of code just below the "hide_copyright" control.

```
$wp_customize->add_setting(
    'logo_placement',
    array(
        'default' => 'left',
    )
);

$wp_customize->add_control(
    'logo_placement',
    array(
        'type' => 'radio',
        'label' => 'Logo placement',
        'section' => 'example_section_one',
        'choices' => array(
            'left' => 'Left',
            'right' => 'Right',
            'center' => 'Center',
        ),
    )
);
```

You'll notice that just like the text input, we've provided a default value in the add_setting() properties just in case none has been specified yet. The real change in radio buttons over the previously described input types is the addition of the "choices" array this array allows us to specify which options should be available. The first part of each associative array element is the name of the radio button element. This is also the name we will use when retrieving the values from the database. The second part of each array element is the label that will be attached to the radio button.

To make use of this new feature in the theme, we will want to add the following code just above the "wp_head" action hook in the header.php file of the theme.

```
<?php
    $example_position = get_theme_mod( 'logo_placement' );
    if( $example_position != '' ) {
        switch ( $example_position ) {
            case 'left':
                // Do nothing. The theme already aligns the
                break;
            case 'right':
                echo '<style type="text/css">';
                echo '#header #logo{ float: right; }';
                echo '</style>';
```

```
                break;
            case 'center':
                echo '<style type="text/css">';
                echo '#header{ text-align: center; }';
                echo '#header #logo{ float: none; margin-lef
                echo '</style>';
                break;
        }
    }
?>
```

This should be pretty straightforward. The code above first gets the logo placement value from the database and assigns it to a variable. Then if the logo position is not blank it runs through the possible choices and writes out the styles to move the logo around. Note that if the logo position is "left", no CSS is created because the theme already displays the logo to the left.

5.3 Select lists

Creating a select list is almost exactly the same as creating radio buttons. To demonstrate, we'll add a select list that will let us replace the default "powered by" footer message with something a little more fun. We'll add the following code right below the radio button setting and control.

```
$wp_customize->add_setting(
    'powered_by',
    array(
        'default' => 'wordpress',
    )
);

$wp_customize->add_control(
    'powered_by',
    array(
        'type' => 'select',
        'label' => 'This site is powered by:',
        'section' => 'example_section_one',
        'choices' => array(
            'wordpress' => 'WordPress',
            'hamsters' => 'Hamsters',
            'jet-fuel' => 'Jet Fuel',
            'nuclear-energy' => 'Nuclear Energy',
        ),
    )
);
```

That was easy! Now we have a new select list in the theme customizer and are ready to use it in the theme. We'll start by removing the old "powered by" message from the footer.php file by deleting these lines:

```
<a href="<?php echo esc_url(__('http://themeid.com/responsive
        <?php printf('Responsive Theme'); ?></a>
```

```
<?php esc_attr_e('powered by', 'responsive'); ?> <a href="<?p
        <?php printf('WordPress'); ?></a>
```

In it's place we'll add this single line of code:

```
This site is powered by <?php echo get_theme_mod( 'powered_by
```

If no "powered by" setting has been saved, the footer will read "This site is powered by WordPress." However, if the setting has been saved, "WordPress" will be replaced by the fun term of choice from the select list.

6. Data Sanitization

I mentioned earlier in this tutorial that for the sake of simplicity, I wouldn't try to throw in data sanitization right at the beginning. However, I don't want to give the impression that I put it off because it's not important. I'm giving it an entire section of this tutorial, right? So yes, I do think it is very important. However, I'm not going to go into all the reasons why it is so important or the consequences of failing to sanitize data. I'll just leave it at this: never trust user input!

6.01 Sanitize a text input

Okay, now we can get to the code. Data sanitization functions are added by using the "sanitize_callback" argument when creating a new setting. Remember the original textbox we added at the beginning of the tutorial for modifying the copyright notice in the footer? This is what it looked like:

```
$wp_customize->add_setting(
    'copyright_textbox',
    array(
        'default' => 'Default copyright text',
    )
```

Now we simply need to add the "sanitize_callback" argument to the arguments array like this:

```
$wp_customize->add_setting(
    'copyright_textbox',
    array(
        'default' => 'Default copyright text',
        'sanitize_callback' => 'example_sanitize_text',
    )
```

Whenever the "copyright_textbox" value is saved, it first calls the "example_sanitise_text" function and passes it the value to be saved. What we want to do is check to make sure that the value is valid before saving it

permanently. In this case, we want to save a simple string. Lets say we don't mind if they use a little HTML in their copyright notice, but we definitely don't want to allow script tags to be added. We also want to make sure that if they do add any HTML tags, they are closed properly. With that in mind, here is the sanitization function we can use to accomplish what we just described:

```
function example_sanitize_text( $input ) {
    return wp_kses_post( force_balance_tags( $input ) );
}
```

The force_balance_tags() function ensures that no tags are left unclosed, while the wp_kses_post() ensures that only safe tags make it into the database (the same tags that are allowed in a standard WordPress post. These functions are built into WordPress and are documented in the WordPress Codex.

6.02 Santize a checkbox

I demonstrated how to add a sanitization callback in the previous section dealing with sanitizing text inputs, so I'm not going to repeat that here. Instead, I'll go straight to the sanitization function that will be used to sanitize a checkbox.

The customizer uses a value of 1 (true) for a checked checkbox and a blank value for an unchecked box. In other words, the only two values our sanitization function should ever save are a 1 or a blank string. Here's what it looks like:

```
function example_sanitize_checkbox( $input ) {
    if ( $input == 1 ) {
        return 1;
    } else {
        return '';
    }
}
```

If the input is a 1 (indicating a checked box) then the function returns a one. If the input is anything else at all, the function returns a blank string. This prevents anything harmful from being saved to the database.

6.03 Sanitize radio buttons or select lists

Sanitizing multiple choice options such as radio buttons and select lists is a little more difficult because each time we use a multiple choice option the valid choices are different. For example, to sanitize the "Logo placement" setting we added earlier, we would want to make sure only input matching "left", "right", or "center" would be accepted. However, our "Powered by" setting should accept "wordpress", "hamsters", "jet-fuel", or "nuclear-energy" as valid instead. For this reason, we'll need to create separate callback functions for each of our multiple

choice fields. I'll mention again that I demonstrated how to add a sanitization callback in the previous section dealing with sanitizing text inputs, so I'm not going to repeat that here. Instead, I'll go straight to the sanitization functions that will be used to sanitize our multiple choice fields.

Let's start with the "Logo placement" radio buttons. To make things simple, we'll use the same array we created when adding the "Logo placement" control in order to check if the input matches our expected input.

```
function example_sanitize_logo_placement( $input ) {
    $valid = array(
        'left' => 'Left',
        'right' => 'Right',
        'center' => 'Center',
    );

    if ( array_key_exists( $input, $valid ) ) {
        return $input;
    } else {
        return '';
    }
}
```

The first half of this sanitization function stores our original logo placement options in an array. The second half of the function checks to see if the input is valid by comparing it to the array keys using the array_key_exists() function provided by PHP. If the input matches one of the array keys, we return the input as is. However, if the input doesn't match an existing array key, it is replaced with a blank string. This ensures that only values matching our original array keys are stored in the database.

The sanitization function for the select list works in exactly the same way. The only difference is that we replace the logo placement array with the array from our "Powered by" select list.

```
function example_sanitize_powered_by( $input ) {
    $valid = array(
        'wordpress' => 'WordPress',
        'hamsters' => 'Hamsters',
        'jet-fuel' => 'Jet Fuel',
        'nuclear-energy' => 'Nuclear Energy',
    );

    if ( array_key_exists( $input, $valid ) ) {
        return $input;
    } else {
        return '';
    }
}
```

...and that's how we sanitize all the standard form inputs provided by the WordPress theme customizer.

7. Special control types

In addition to the general control types covered earlier, WordPress offers a number of special control types. Since we've already covered the use of the get_theme_mod() function to retrieve saved customizer settings from the database, I'm not going to prolong this tutorial by trying to come up with a use case for every single control type. I'll simply show you how to create the controls and let you decide how you want to use the saved values in your theme.

7.01 Drop down page list

Creating a drop down page list is quite simple. We can start with the same code we would use to create a checkbox and simply change the setting name, the sanitization callback, and the control type.

```
$wp_customize->add_setting(
    'page-setting',
    array(
        'sanitize_callback' => 'example_sanitize_integer',
    )
);

$wp_customize->add_control(
    'page-setting',
    array(
        'type' => 'dropdown-pages',
        'label' => 'Choose a page:',
        'section' => 'example_section_one',
    )
);
```

That was easy! Notice the "type" is set to "dropdown-pages" which will automatically create a select list in the theme customizer allowing us to choose a page. The value stored in the database for this setting is the page ID. Because of this, we want to make sure that the input data is an integer before saving it to the database. Here is the sanitization function we can use to do that.

```
function example_sanitize_integer( $input ) {
    if( is_numeric( $input ) ) {
        return intval( $input );
    }
}
```

7.02 Color picker

This is where we start to get into the more interesting theme customizer controls. Adding the setting itself hasn't changed, but adding the control is a bit

different. Let's start by adding the setting.

```
$wp_customize->add_setting(
    'color-setting',
    array(
        'default' => '#000000',
        'sanitize_callback' => 'sanitize_hex_color',
    )
);
```

Looks familiar, right? The only difference is the sanitization callback function we're using. WordPress has a built-in function for sanitizing hexadecimal color codes, which is the exact type of output our color picker will create. We also set the default color to black (#000000).

Now lets take a look at the code used to create the color picker control.

```
$wp_customize->add_control(
    new WP_Customize_Color_Control(
        $wp_customize,
        'color-setting',
        array(
            'label' => 'Color Setting',
            'section' => 'example_section_one',
            'settings' => 'color-setting',
        )
    )
);
```

The color picker control was built into WordPress using object oriented PHP, so the format of the add_control() call is different than the previous calls we've made. Notice the "new WP_Customize_Color_Control" line. This tells WordPress we want a new instance of the WP_Customize_Color_Control class. If you don't know what that means, don't worry about it. You don't need to understand object oriented programming to use the theme customizer, so I'm not going to try to explain it here.

The WP_Customize_Color_Control accepts three properties. The first, $wp_customize, is always the same, so I'm not going to try to offer an explanation. Just make sure to use it. The second is the setting ID. This must be the same as the ID used within the add_setting() call. The third is the arguments array, explained in the following table:

Array: $args		
label	string	Label/title for this setting.
section	string	The section to which this setting should be added.
setttings	string	The setting ID used earlier in the add_control() call as well as the add_setting() call.

And there it is. WordPress does the rest of the work of creating the color picker us.

7.03 File upload

There is another class built into WordPress that will allow us add a file upload setting to the theme customizer. WordPress itself limits the file types that can be uploaded, so I'm going to leave out the data sanitization function this time.

```
$wp_customize->add_setting( 'file-upload' );

$wp_customize->add_control(
    new WP_Customize_Upload_Control(
        $wp_customize,
        'file-upload',
        array(
            'label' => 'File Upload',
            'section' => 'example_section_one',
            'settings' => 'file-upload'
        )
    )
);
```

Just like the color picker, the file upload control is displayed by instantiating a new instance of a class. This time, the class in named WP_Customize_Upload_Control. Thankfully, it accepts the same arguments as the color so it's pretty straight forward. All we'll need to change is the setting ID.

7.04 Image upload

The image upload class is an extension of the file upload class and is called in the same way. We just change WP_Customize_Upload_Control to WP_Customize_Image_Control and use a new setting ID.

```
$wp_customize->add_setting( 'img-upload' );

$wp_customize->add_control(
    new WP_Customize_Image_Control(
        $wp_customize,
        'img-upload',
        array(
            'label' => 'Image Upload',
            'section' => 'example_section_one',
            'settings' => 'img-upload'
        )
    )
);
```

Once again, WordPress automatically limits the types of files that can be uploaded, but it isn't limited to just images. You may want to create a sanitization

function to make sure no other file types are uploaded using the image upload control. However, since this is a tutorial on the theme customizer, not on data sanitization, I'm not going to go into that here.

8. Building Custom Controls

Now that we've covered the controls built into WordPress, how do we go about building additional controls? Otto wrote a great post on making custom controls, and this section will draw heavily on his tutorial.

You might have noticed earlier that the textarea input was missing from the list of control types built into WordPress. Here we will create a custom control to facilitate the use of textareas in the theme customizer.

To create our own control, we'll need to extend the WP_Customize_Control class and override the render_content() function to output our new control. If you're familiar with object oriented PHP, this will make sense to you, and if you're not familiar with it, well, just follow along. It won't be hard.

We want to place our class inside the example_customizer() function we created at the beginning of this tutorial. This is the function we used to setup the customizer and add all our settings. In order for the class to work properly, it must be defined somewhere inside this function:

```
/**
 * Adds the individual sections, settings, and controls to th
 */
function example_customizer( $wp_customize ) {

    ...

}
add_action( 'customize_register', 'example_customizer' );
```

Please note: there is another way to declare the class outside this function, but I'm not going to cover that here.

To create our new textarea class, we'll place the following section of code into the function we just mentioned:

```
/**
 * Adds textarea support to the theme customizer
 */
class Example_Customize_Textarea_Control extends WP_Customiz
    public $type = 'textarea';

    public function render_content() {
        ?>
            <label>
```

```
                <span class="customize-control-title"><?php
                <textarea rows="5" style="width:100%;" <?php
            </label>
        <?php
    }
}
```

Notice the render_content() function displays our textarea, along with the label for this setting. The WP_Customize_Control class that we're extending does the rest of the work for us.

Now we can add a setting to our customizer that makes use of the textarea class we just created. For simplicity sake, I'm not going to add things like the display priority, callback function, etc. that were covered earlier. The code itself is almost exactly the same as the code we used for the upload and upload image controls.

```
$wp_customize->add_setting( 'textarea' );

$wp_customize->add_control(
    new Example_Customize_Textarea_Control(
        $wp_customize,
        'textarea',
        array(
            'label' => 'Textarea',
            'section' => 'example_section_one',
            'settings' => 'textarea'
        )
    )
);
```

The only changes are:

1. The setting ID has been changed to "textarea"
2. The name of the class being called has been changed to "Example_Customize_Textarea_Control"
3. The label has been changed to "Textarea"

This should give you a good starting point to create your own classes to extend the theme customizer.

9. Adding New Settings to Existing Sections

Maybe you want to add a new setting to a section that already exists. For example, because the Responsive theme supports the WordPress custom background feature, the "Colors" settings section is automatically created and the "Background Color" settings is placed inside. Let's add another option to the Colors section and call it "Font Color." First, we need to find out what the section ID is, so that we can attach our new setting to it. Sections added automatically by

WordPress itself are defined in the **wp-includes/class-wp-customize-manager.php** file. After looking through that file, we find the "colors" is the ID of the section titled "Colors."

Since we already created a color option earlier, we can just copy/paste the code from our previous color option and make a few changes.

```
$wp_customize->add_setting(
    'font-color',
    array(
        'default' => '#444444',
        'sanitize_callback' => 'sanitize_hex_color',
    )
);

$wp_customize->add_control(
    new WP_Customize_Color_Control(
        $wp_customize,
        'font-color',
        array(
            'label' => 'Font Color',
            'section' => 'colors',
            'settings' => 'font-color'
        )
    )
);
```

Here are the changes:

1. Change the previous setting ID of "color-setting" to "font-color" in all three places it appears.
2. Change the default color.
3. Change the label.
4. And finally, set the section property to "color" to make sure it is added to the existing "Colors" section.

We've already covered how to use the saved settings in the theme, so I'm going to leave it to you to apply the saved font color to the theme.

10. Using AJAX to update the live preview

Last but not least, let's return to the "postMessage" transport option we saw briefly in section three. We'll add a new option to change the background color of the featured content area on the home page. First, we'll add another color picker to the "Colors" section of the theme customizer like this:

```
$wp_customize->add_setting(
    'featured-background',
```

```
    array(
        'default' => '#ffffff',
        'sanitize_callback' => 'sanitize_hex_color',
        'transport' => 'postMessage',
    )
);

$wp_customize->add_control(
    new WP_Customize_Color_Control(
        $wp_customize,
        'featured-background',
        array(
            'label' => 'Featured Background',
            'section' => 'colors',
            'settings' => 'featured-background'
        )
    )
);
```

The only difference between this code and the code we used to create color pickers earlier is the label/id and the addition of the "transport" argument. Now that we have the "transport" method set, we need to write a function to handle the AJAX updates. This function will be hooked to the wp_footer action hook. To accomplish this, we must place the following code inside the example_customizer() function we created way back at the beginning:

```
if ( $wp_customize->is_preview() ) {
    add_action( 'wp_footer', 'example_customize_preview', 21)
}
```

This code checks to make sure that the theme customizer is being used. If it is, the example_customize_preview() function is hooked to wp_footer. The final argument (21) is just the priority that the function is given. It must be set to at least 20 in order to work properly.

Finally, we want to create the example_customize_preview() function to add the Javascript that will handle the AJAX update.

```
function example_customize_preview() {
    ?>
    <script type="text/javascript">
        ( function( $ ) {
            wp.customize('featured-background',function( val
                value.bind(function(to) {
                    $('#featured').css('background-color', t
                });
            });
        } )( jQuery )
    </script>
    <?php
}  // End function example_customize_preview()
```

This function simply adds a small piece of javascript to the footer of the theme (and only when the theme customizer is being used). This function can serve as a starting point for handling almost any type of customization data passed in via AJAX. The code above does basically two things. First, it gets the value of the customization field whenever it changes (in this case, it is the "featured-background" value as you can see from line five). Second, it does something with that value. Since we're using a color input, we set the modified value of the input to be the background color of the #featured element (line 7) by using the css() jQuery function.

11. Using AJAX with Preexisting Settings

But what if we want to modify the transport setting of a built in setting? Thankfully, there's a way to do that too. To demonstrate, we'll be changing the built in "Site Title" option so that it will pass its modified value to the theme customizer using "postMessage" rather than requiring a refresh. First we want to add the following line to the main example_customizer() function created earlier:

```
1  $wp_customize->get_setting('blogname')->transport='postMessag
```

This ensures that changes to the setting value will be passed to the live preview via AJAX. However, in order for it to work properly, we need a Javascript function to handle the value when it's passed to the live preview. We can add this code to the example_customize_preview() function we created in the previous step. It simply updates the anchor text inside the ".site-name" element whenever the value changes.

```
1  wp.customize('blogname',function( value ) {
2      value.bind(function(to) {
3          $('.site-name a').html(to);
4      });
5  });
```

Please note: The Responsive theme by Emil Uzelac that has been used throughout this tutorial doesn't display the site name if a header image is present. Make sure to disable the header image if you're having trouble testing the code on this theme.

12. More?

Did I miss anything? If you have any suggestions on improving this developer's guide, or even if you just found it helpful and want to let me know, please don't hesitate to contact me via twitter. Thanks!

Sources/Further reading (all from Otto):

- WordPress source code.
- How to leverage the Theme Customizer in your own themes
- Theme Customizer Part Deux: Getting rid of Options pages
- Making a custom control for the Theme Customizer

« A Beginning

Theme Toolkit Tutorial: The WordPress Theme Customizer »