

Day 3

Create Theme

**BY:**

**BRUNO ANGELO MEDEIROS**bruno.medeiros@sc.senai.br

# CREATING THE THEME

Create a new directory for our theme, we can called **startwordpress**.

A WordPress theme needs only two files to exist: **style.css** and **index.php**.

## STYLE.CSS

In our custom theme folder, create **style.css**. It simply contains a comment that alerts WordPress that a theme exists here. Change the name, author, description, and so on.

1. /\*\*
2. \* Theme Name: Start WordPress
3. \* Author: Bruno Medeiros
4. \* Description: Bootstrap Blog template converted to WordPress
5. \* Version: 0.0.1
6. \* Tags: bootstrap
7. \*/

## INDEX.PHP – HOME

The index file controls what the homepage of our WordPress theme looks like. By default it is a loop that queries and then displays the most recent blog posts, with a link in the bottom to view previous posts.

Alternately, we can specify in wp-admin -> settings -> reading to have the home page be a page we created ourself in WordPress. In that case, we specify a different page/URL for the regular blog posts to appear on, and that page is generated by index.php.

Using the template provided, we need move **index.html** to our custom theme folder and copy the the **style.css** from **css** folder to our **style.css** created previously. After that the copy the others folders – js, fonts, images – in our custom theme folder.

Now, activate the theme and go back to our main URL. Of course, it doesn’t do anything yet beyond what a static HTML site can do.

Locate where we linked to the CSS stylesheet in the head of **index.php**, and replace the above code with the below code.

1. **<link** href="<?= get\_bloginfo('template\_directory'); ?>/style.css" rel="stylesheet"**>**

If this we tell it to dynamically link css to the themes folder. The concept will be the same for images, javascript, and most other files we have in the themes folder.

## DIVIDING OUR PAGE INTO SECTIONS

Right now, everything is in **index.php**. But obviously we want the header, footer and sidebar on all the pages to be the same.

We’re going to divide **index.php** into four sections – header.php, footer.php, sidebar.php and content.php.

## HEADER.PHP

Global file that displays headers and navigation, also contains HTML code.

The header usually contains all the necessary head styles and the top navigation to the website.

Everything from **<!DOCTYPE html>** to the main blog header will be in the header file. The only addition we will make to the code is adding **<?php wp\_head(); ?>** right before the closing **</head>.**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title><?php** echo get\_bloginfo( 'name' ); **?></title>**
5. **<link** href="<?= get\_bloginfo('template\_directory'); ?>/css/bootstrap.css" rel='stylesheet' type='text/css' **/>**
6. **<link** href="<?= get\_bloginfo('template\_directory'); ?>/style.css" rel='stylesheet' type='text/css' **/>**
7. **<meta** name="viewport" content="width=device-width, initial-scale=1"**>**
8. **<meta** http-equiv="Content-Type" content="text/html; charset=utf-8" **/>**
9. **<meta** name="keywords" content="Personal Blog Responsive web template, Bootstrap Web Templates, Flat Web Templates, Andriod Compatible web template,
10. Smartphone Compatible web template, free webdesigns for Nokia, Samsung, LG, SonyErricsson, Motorola web design"
11. **/>**
12. **<script** type="application/x-javascript"**>** addEventListener("load", function() { setTimeout(hideURLbar, 0); }, false); function hideURLbar(){ window.scrollTo(0,1); } **</script>**
13. <!----webfonts---->
14. **<link** href='http://fonts.googleapis.com/css?family=Oswald:100,400,300,700' rel='stylesheet' type='text/css'**>**
15. **<link** href='http://fonts.googleapis.com/css?family=Lato:100,300,400,700,900,300italic' rel='stylesheet' type='text/css'**>**
16. <!----//webfonts---->
17. **<script** src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"**></script>**
18. <!--end slider -->
19. <!--script-->
20. **<script** type="text/javascript" src="<?= get\_bloginfo('template\_directory'); ?>/js/move-top.js"**></script>**
21. **<script** type="text/javascript" src="<?= get\_bloginfo('template\_directory'); ?>/js/easing.js"**></script>**
22. **<?php** wp\_head(); **?>**
23. **</head>**
24. **<body>**
25. <!---header---->
26. **<div** class="header"**>**
27. **<div** class="container"**>**
28. **<div** class="logo"**>**
29. **<a** href="<?= get\_bloginfo( 'wpurl' );?>"**><img** src="<?= get\_bloginfo('template\_directory'); ?>/images/logo.jpg" title="" **/></a>**
30. **</div>**
31. <!---start-top-nav---->
32. **<div** class="top-menu"**>**
33. **<span** class="menu"**>** **</span>**
34. **<ul>**
35. **<li** class="active"**><a** href=" <?= get\_bloginfo( 'wpurl' );?>"**>**HOME**</a></li>**
36. **<li><a** href="about.html"**>**ABOUT**</a></li>**
37. **<li><a** href="contact.html"**>**CONTACT**</a></li>**
38. **<div** class="clearfix"**>** **</div>**
39. **</ul>**
40. **</div>**
41. **<div** class="clearfix"**></div>**
42. <!---//End-top-nav---->
43. **</div>**
44. **</div>**
45. <!--/header-->

## FOOTER.PHP

Contains instructions for global footer and closes HTML tags.

Same thing for the footer as the header. It will include whatever visible footer we have, our JS links if we’ve any and **<?php wp\_footer(); ?>** right before **</body>**.

1. <!--footer-starts-->
2. **<div** class="footer"**>**
3. **<div** class="container"**>**
4. **<p>**Copyrights © 2015 Blog All rights reserved | Template by **<a** href="http://w3layouts.com/"**>**W3layouts**</a></p>**
5. **</div>**
6. **</div>**    <!--footer-end-->
7. **<?php** wp\_footer(); **?>**
8. **</body>**
9. **</html>**

## SIDEBAR.PHP

Sidebar display is controlled in this file. Multiple sidebars can be set up in **functions.php** and contents of sidebar widgets are set up from the WordPress wp-admin panel.

Most websites, especially blogs, will have a side area for including content such as archives, tags, categories, ads, and etc.

## CONTENT.PHP

If the sidebar is where all the secondary information goes, the content is where all the articles and main content of the website go.

## INDEX

Now we’re going to add everything back in. Here’s our new index.php.

1. **<?php** get\_header(); **?>**
3. **<div** class="content"**>**
4. **<div** class="container"**>**
5. **<div** class="content-grids"**>**
6. **<div** class="col-md-8 content-main"**>**
8. **<?php**
9. get\_template\_part( 'content', get\_post\_format() );
10. **?>**
12. **</div>**
13. **<?php** get\_sidebar(); **?>**
14. **</div>**
15. **</div>**
16. **</div>**
18. **<?php** get\_footer(); **?>**

# MAIN SETTINGS

Before we start pulling in posts and pages, we need to configure some main settings of WordPress. In our dashboard, go to Settings > General. Set our title.

Now, we need to configure some main settings of WordPress. For example, using a HTML template we need the <title> and <h1> of our site to be changeable through the back end.

In **header.php** and change the contents of the title tag to this code:

1. <?php echo get\_bloginfo( 'name' ); ?>

And if theme has description add this one.

1. <?php echo get\_bloginfo( 'description' ); ?>

Or simply remove the title tag from our **header.php** entirely, and in **functions.php**, add this code block:

1. // WordPress Titles
2. add\_theme\_support( 'title-tag' );

Finally, we want the title or logo to always take us back to the main page, **bloginfo('wpurl');** is the code that will do that.

## THE LOOP

It’s the most important function of WordPress. All of our content is generated through a loop.

The loop starts with a query (which determines which posts or pages to grab), and ends with a PHP “endwhile” statement. We can specify the output of titles, post content, metadata, custom fields and commenting all within the loop and each element is output for each post or page until the query is done. We can set up multiple loops and queries on a single page;

A simple example of a post loop:

1. <?php **if** ( have\_posts() ) : **while** ( have\_posts() ) : the\_post(); ?>
3. <!-- contents of the loop -->
5. <?php **endwhile**; **endif**; ?>

Here’s our new index file.

1. **<?php** get\_header(); **?>**
3. **<div** class="content"**>**
4. **<div** class="container"**>**
5. **<div** class="content-grids"**>**
6. **<div** class="col-md-8 content-main"**>**
8. **<?php**
9. if ( have\_posts() ) : while ( have\_posts() ) : the\_post();
11. get\_template\_part( 'content', get\_post\_format() );
13. endwhile; endif;
14. **?>**
16. **</div>**
17. **<?php** get\_sidebar(); **?>**
18. **</div>**
19. **</div>**
20. **</div>**
22. **<?php** get\_footer(); **?>**

The only thing inside our loop is **content.php**, which will contain the contents of one single post. So open **content.php** and change the contents to this:

1. **<div** class="content-grid"**>**
2. **<div** class="content-grid-info"**>**
3. **<img** src="<?= get\_bloginfo('template\_directory'); ?>/images/post1.jpg" alt=""**/>**
4. **<div** class="post-info"**>**
5. **<h4><a** href="single.html"**><?php** the\_title(); **?></a><?php** the\_author(); **?>** on **<?php** the\_date(); **?> </h4>**
6. **<?php** the\_content(); **?>**
7. **<a** href="single.html"**><span></span>**READ MORE**</a>**
8. **</div>**
9. **</div>**
10. **</div>**

Let’s make the sidebar dynamic, as well. There should be a description and archive list in the sidebar. Delete all the <li>s under Archives and change it to this code.

1. **<?php** wp\_get\_archives( 'type=monthly' ); **?>**

## PAGE.PHP – INDIVIDUAL PAGES

Page.php controls what pages look like. We can choose to eliminate sidebars or other elements, add other unique elements for pages alone.

First, we’re going to edit the navbar so that the links lead to the pages. Back in **header.php**, find and change this code.

1. **<ul>**
2. **<li** class="active"**><a** href=" <?= get\_bloginfo( 'wpurl' );?>"**>**HOME**</a></li>**
3. **<?php** wp\_list\_pages( '&title\_li=' ); **?>**
4. **<div** class="clearfix"**>** **</div>**
5. **</ul>**

To list all added pages, we use the function **wp\_list\_pages()**,where will list all the pages we have in an unordered list. **'title\_li='** is telling the code not to add a “Pages” title before the list.

WordPress also allows we to create different page templates within our WordPress theme for different types of pages.

To create pages to have a different layout than the blog posts without sidebars for example. We’ll create **page.php**, which will be very similar to the **index** except have a full 12-wide grid instead of an 8-wide content and 4-wide sidebar.

1. **<?php** get\_header(); **?>**
3. **<div** class="content"**>**
4. **<div** class="container"**>**
5. **<div** class="content-grids"**>**
6. **<div** class="col-md-12"**>**
8. **<?php**
9. if ( have\_posts() ) : while ( have\_posts() ) : the\_post();
11. get\_template\_part( 'content', get\_post\_format() );
13. endwhile; endif;
14. **?>**
16. **</div>**
17. **</div>**
18. **</div>**
19. **</div>**
21. **<?php** get\_footer(); **?>**

## SINGLE.PHP – INDIVIDUAL POSTS

The display of individual posts in our WordPress theme is controlled by a little file called **single.php**. It contains a loop that queries just one post and displays it. We can specify if we want sidebars, if we want it to look different than the other pages on the site.

In this case we’ll need change some classes and html structure and duplicate of **the loop** in the **index.php**, except we’re going to change **'content'** to **'content-single'.**

1. <?php get\_header(); ?>
3. <div **class**="single">
4. <div **class**="container">
5. <div **class**="col-md-8 single-main">
6. <?php
7. **if** ( have\_posts() ) : **while** ( have\_posts() ) : the\_post();
9. get\_template\_part( 'content-single', get\_post\_format() );
11. **endwhile**; **endif**;
12. ?>
13. </div>
14. <?php get\_sidebar(); ?>
15. </div>
16. </div>
18. <?php get\_footer(); ?>

Now we’ll create **content-single.php**. So now we can see that **index.php** is pulling in **content.php**, and **single.php** is pulling in **content-single.php**.

1. <div **class**="single-grid">
2. <img src="<?= get\_bloginfo('template\_directory'); ?>/images/post1.jpg" alt=""/>
3. <?php the\_content(); ?>
4. </div>
5. <ul **class**="comment-list">
6. <h5 **class**="post-author\_head">Written by <a href="#" title="Posts by <?php the\_author(); ?>" rel="author"><?php the\_author(); ?></a></h5>
7. <li>
8. <img src="<?= get\_bloginfo('template\_directory'); ?>/images/avatar.png" **class**="img-responsive" alt="">
9. <div **class**="desc">
10. <p>View all posts by: <a href="#" title="Posts by <?php the\_author(); ?>" rel="author"><?php the\_author(); ?></a></p>
11. </div>
12. <div **class**="clearfix"></div>
13. </li>
14. </ul>

The **the\_permalink()** function is used to reference the post in the list in **single.php**, we’re going to link to the single page. Going back to the original **content.php**, we have the title of each article.

1. **<div** class="content-grid"**>**
2. **<div** class="content-grid-info"**>**
3. **<img** src="<?= get\_bloginfo('template\_directory'); ?>/images/post1.jpg" alt=""**/>**
4. **<div** class="post-info"**>**
5. **<h4><a** href="<?php the\_permalink(); ?>"**><?php** the\_title(); **?></a><?php** the\_author(); **?>** on **<?php** the\_date(); **?></h4>**
6. **<?php** the\_content(); **?>**
7. **<a** href="<?php the\_permalink(); ?>"**><span></span>**READ MORE**</a>**
8. **</div>**
9. **</div>**
10. **</div>**

Finally, we’ll want to change **the\_content()** to **the\_excerpt()** on **content.php**. The excerpt will only show the first 55 words of our post, instead of the entire contents.

1. **<div** class="content-grid"**>**
2. **<div** class="content-grid-info"**>**
3. **<img** src="<?= get\_bloginfo('template\_directory'); ?>/images/post1.jpg" alt=""**/>**
4. **<div** class="post-info"**>**
5. **<h4><a** href="<?php the\_permalink(); ?>"**><?php** the\_title(); **?></a>**by **<?php** the\_author(); **?>** on **<?php** the\_date(); **?></h4>**
6. **<?php** the\_excerpt(); **?>**
7. **<a** href="<?php the\_permalink(); ?>"**><span></span>**READ MORE**</a>**
8. **</div>**
9. **</div>**
10. **</div>**

### PAGINATION

We’ll can notice, the loop has **if** and **while**, then later **endif** and **endwhile**. To insert pagination, we’ll have to put it after the **endwhile** but before the **endif**. This means that it won’t repeat for each loop, but will only show up once based on posts.

Pagination links are called like this:

1. <?php next\_posts\_link( 'Older posts' ); ?>
2. <?php previous\_posts\_link( 'Newer posts' ); ?>

By default, 10 posts will show up on a page before it will link to another page. Go to **index.php** and add the code above, the result is like this:

1. **<?php** get\_header(); **?>**
3. **<div** class="content"**>**
4. **<div** class="container"**>**
5. **<div** class="content-grids"**>**
6. **<div** class="col-md-8 content-main"**>**
8. **<?php**
9. if ( have\_posts() ) : while ( have\_posts() ) : the\_post();
11. get\_template\_part( 'content', get\_post\_format() );
13. endwhile; **?>**
15. **<?php** next\_posts\_link( 'Older posts' ); **?>**
16. **<?php** previous\_posts\_link( 'Newer posts' ); **?>**
18. **<?php** endif; **?>**
20. **</div>**
21. **<?php** get\_sidebar(); **?>**
22. **</div>**
23. **</div>**
24. **</div>**
26. **<?php** get\_footer(); **?>**

### COMMENTS.PHP

This controls the output of comments, which can be included in the loop if we desire comments on our theme.

First, we’re going to go back to **single.php** and enable the comments.

1. **if** ( have\_posts() ) : **while** ( have\_posts() ) : the\_post();
3. get\_template\_part( 'content-single', get\_post\_format() );
5. **if** (comments\_open() || get\_comments\_number()):
7. comments\_template();
9. **endif**;
11. **endwhile**; **endif**;

This is just telling the single post to display the comments template. Now we’ll create **comments.php.**

1. **<?php** if ( post\_password\_required() ) {
2. return;
3. } **?>**
4. **<div** id="comments" class="comments-area"**>**
5. **<?php** if ( have\_comments() ) : **?>**
6. **<h3** class="comments-title"**>**
7. **<?php**
8. printf( \_nx( 'One comment on “%2$s”', '%1$s comments on “%2$s”', get\_comments\_number(), 'comments title'),
9. number\_format\_i18n( get\_comments\_number() ), get\_the\_title() );
10. **?>**
11. **</h3>**
12. **<ul** class="comment-list"**>**
13. **<?php**
14. wp\_list\_comments( array(
15. 'short\_ping'  =**>** true,
16. 'avatar\_size' =**>** 50,
17. ) );
18. **?>**
19. **</ul>**
20. **<?php** endif; **?>**
21. **<?php** if ( ! comments\_open() && get\_comments\_number() && post\_type\_supports( get\_post\_type(), 'comments' ) ) : **?>**
22. **<p** class="no-comments"**>**
23. **<?php** \_e( 'Comments are closed.' ); **?>**
24. **</p>**
25. **<?php** endif; **?>**
26. **<div** class="content-form"**>**
27. **<?php** comment\_form(); **?>**
28. **</div>**
29. **</div>**

First, we’re setting functionality to prevent users from posting comments if we’ve set our settings to password protected comments (**post\_password\_required()**). Then we’re creating a comments div, and if there are comments (**have\_comments()**), it will display how many comments there are on the post (**get\_comments\_number()),** followed by the list of comments (**wp\_list\_comments()**). If the comments are closed **(! comments\_open()**), it will let we know; at the end will be the form to submit a comment (**comment\_form()**).

## ARCHIVE.PHP, CATEGORY.PHP, TAG.PHP – ARCHIVES

We can control the look and feel of different archives using template files also. If there is no archive file, the archives will look like index.php; however we can create an archive.php to override that. If we create a file called category.php, it will override archive.php for categories only. If we create a tag.php, we can override it for tag archives only.

## FUNCTIONS.PHP

Located in our theme directory, **functions.php** allows we to put our own custom PHP code in order to modify core elements of our theme. It is often used to specify multiple sidebars, change the number of characters in the excerpt or add custom admin panel options for wp-admin.

Plugins and custom functions are basically the same, any code we create can be made into a plugin, and vice versa. The only difference is that anything we place in our theme’s functions is only applied while that theme is actively selected.

Like we learn in the others days, now we’ll delete the links to the stylesheets and scripts that we have in our header and footer. They’re no longer going to be hard coded into the theme.

Now here’s the first code block we’re going to put in **functions.php**:

1. // Add scripts and stylesheets
2. **function** startwordpress\_scripts() {
3. wp\_enqueue\_style( 'bootstrap', get\_template\_directory\_uri() . '/css/bootstrap.css', **array**());
4. wp\_enqueue\_style( 'style', get\_template\_directory\_uri() . '/style.css', **array**());
5. wp\_enqueue\_style( 'Oswald', '//fonts.googleapis.com/css?family=Oswald:100,400,300,700' );
6. wp\_enqueue\_style( 'Lato', '//fonts.googleapis.com/css?family=Lato:100,300,400,700,900,300italic' );
8. wp\_enqueue\_script( 'jquery', '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js', **array**(), '');
9. wp\_enqueue\_script( 'move-top', get\_template\_directory\_uri() . '/js/move-top.js', **array**( 'jquery' ), '');
10. wp\_enqueue\_script( 'easing', get\_template\_directory\_uri() . '/js/easing.js', **array**( 'jquery' ), '');
11. }
13. add\_action( 'wp\_enqueue\_scripts', 'startwordpress\_scripts' );

# LEARN THE APIs

## OPTIONS

It's the standardized way of storing and retrieving our theme's options. While we don't see too much of it, it works together with the Settings API, so it's important that we learn it well.

## SETTING

This API is the backbone of our "Theme Options" panel. We can choose to write our own panel or search for options panel frameworks, but we will definitely need to study how the API works. If we don't, we might wonder why there are some weird curly braces inside the option we retrieved, or do it all wrong and save every theme option in another database table row.

### THEME SETTINGS PAGE

#### CREATING A MENU ITEM

First we have to create a menu item on the admin panel that will access our theme settings page.

We can create a menu item using the WordPress Menu API. Here’s the code in **functions.php** file:

1. **function** theme\_settings\_page()
2. {
3. ?>
4. <div **class**="wrap">
5. <h1>Theme Panel</h1>
6. <form method="post" action="options.php" enctype="multipart/form-data">
7. <?php
8. settings\_fields("section");
9. do\_settings\_sections("theme-options");
10. submit\_button();
11. ?>
12. </form>
13. </div>
14. <?php
15. }
17. **function** add\_theme\_menu\_item()
18. {
19. add\_menu\_page("Theme Panel", "Theme Panel", "manage\_options", "theme-panel", "theme\_settings\_page", null, 99);
20. }
22. add\_action("admin\_menu", "add\_theme\_menu\_item");

Note: we’re using an **action** to do this. Whenever WordPress sees the **admin\_menu** hook it executes all functions tied to it. We happened to add a function to it ourselves so it will take that into account when building the menu.

We use **add\_menu\_page()** to create a top-level menu entry. This function takes a number of arguments:

1. **Page title** – used in the title tag of the page (shown in the browser bar) when it is displayed.
2. **Menu title** – used in the menu on the left.
3. **Capability** – the user level allowed to access the page.
4. **Menu slug** – the slug used for the page in the URL.
5. **Function** – the name of the function we will be using to output the content of the page.
6. **Icon** – A url to an image or a Dashicons string.
7. **Position** – The position of our item within the whole menu.

And we’re registering a section using **settings\_field** with an ID **section**, **theme-options** is a group ID to all of the fields belonging to a section, the **submit\_button()** function echoes the submit button for our theme settings page, **theme-panel** is the unique ID representing our menu item. Finally, **theme\_settings\_page** is the callback for displaying the content of the page created by the Menu API.

#### SOCIAL NETWORK URLS

To add fields in our settings page to store our Facebook and Twitter profile URLs.

The code to add input text fields is:

1. **function** display\_twitter\_element()
2. {
3. ?>
4. <input type="text" name="twitter\_url" id="twitter\_url" value="<?php echo get\_option('twitter\_url'); ?>" />
5. <?php
6. }
8. **function** display\_facebook\_element()
9. {
10. ?>
11. <input type="text" name="facebook\_url" id="facebook\_url" value="<?php echo get\_option('facebook\_url'); ?>" />
12. <?php
13. }
15. **function** display\_theme\_panel\_fields()
16. {
17. add\_settings\_section("section", "All Settings", null, "theme-options");
19. add\_settings\_field("twitter\_url", "Twitter Profile Url", "display\_twitter\_element", "theme-options", "section");
20. add\_settings\_field("facebook\_url", "Facebook Profile Url", "display\_facebook\_element", "theme-options", "section");
22. register\_setting("section", "twitter\_url");
23. register\_setting("section", "facebook\_url");
24. }
26. add\_action("admin\_init", "display\_theme\_panel\_fields");

After the admin panel is initialized, we’re registering the sections and fields display callbacks. Here we’re using three important functions:

1. **add\_settings\_section** is used to display the section heading and description.
2. **add\_settings\_field** is used to display the HTML code of the fields.
3. **register\_setting** is called to automate saving the values of the fields.

#### UPLOAD LOGO

The **register\_setting** function takes a third argument which is a callback, this callback is fired before it saves the settings into the database. We can utilise this callback to handle uploads.

Here is code to upload a logo in our settings page.

1. **function** logo\_display()
2. {
3. ?>
4. <input type="file" name="logo" />
5. <?php echo get\_option('logo'); ?>
6. <?php
7. }
9. **function** handle\_logo\_upload()
10. {
11. **if**(!**empty**($\_FILES["logo"]["tmp\_name"]))
12. {
13. $urls = wp\_handle\_upload($\_FILES["logo"], **array**('test\_form' => FALSE));
14. $temp = $urls["url"];
15. **return** $temp;
16. }
18. **return** $option;
19. }
21. **function** display\_theme\_panel\_fields()
22. {
23. add\_settings\_section("section", "All Settings", null, "theme-options");
25. add\_settings\_field("logo", "Logo", "logo\_display", "theme-options", "section");
27. register\_setting("section", "logo", "handle\_logo\_upload");
28. }
30. add\_action("admin\_init", "display\_theme\_panel\_fields");

#### RETRIEVING SETTINGS

A theme needs to retrieve the setting values on the front end. The Settings API internally stores the values using the Options API. Therefore, we can retrieve the values using **get\_option()** function.

Here’s the code:

1. <?php
2. $facebook\_url = get\_option('facebook\_url');
3. $twitter\_url = get\_option('twitter\_url');

## THEME CUSTOMIZATION

If we want to give our users the liberty of previewing the theme customizations. It allows the users to change the look and feel of our theme and see the changes in real-time (or sometimes, with a Preview button).

### GETTING STARTED

Let’s begin by laying the groundwork for our customization efforts. We’ll create a **customizer.php** file and make sure to require it in the **functions.php** file.

1. **include**('customizer.php');

The Customizer file will begin with a hooked action that contains all our sections, settings and controls.

1. add\_action( 'customize\_register', 'cd\_customizer\_settings' );
2. **function** cd\_customizer\_settings( $wp\_customize ) {
4. }

### SECTIONS, SETTINGS AND CONTROLS

**Sections** represent the navigation within the Customizer. We should already see four of them: Site Identity, Menus, Static Front Page and Additional CSS. By defining a section we can create a new entry within the navigation and fill it up with controls.

**Controls** are the visual elements, the user interface that allow us to manipulate settings. These may be input fields, text areas, color selectors and other types of controls that serve to create a better user experience.

**Settings** represent the data that we want our theme to accept and use. We create controls to allow users to manipulate settings easily.

#### CREATING A SECTION

To add a new section we’ll use the **$wp\_customize->add\_section()** method. It takes two arguments: a section identifier and an array of additional options like the visible section title and its position within the section list.

1. $wp\_customize->add\_section( 'cd\_colors' , **array**(
2. 'title'      => 'Colors',
3. 'priority'   => 30,
4. ) );

#### CREATING A SETTING

Settings inform WordPress that our theme will be using a value that can be modified by the user. They are added using the **$wp\_customize->add\_setting()** method that takes two parameters: the identifier for the setting and an array of options that contain information such as the default value.

1. $wp\_customize->add\_setting( 'background\_color' , **array**(
2. 'default'     => '#43C6E4',
3. 'transport'   => 'refresh',
4. ) );

The transport actually defaults to refresh, refresh means that when the setting is modified, WordPress should refresh the view. More sophisticated Customizer implementations allow for live previews that only modify the affected element as opposed to the whole page.

#### CREATING A CONTROL

Controls tie settings to user input. They are added **with $wp\_customize->add\_control()** method.

If the first argument is a control object then only one argument is required, an instance of the object. The object will likely have its own additional arguments.

The first argument can also be an identifier, in which case a second argument of options is required.

In our case we’ll be using the WP\_Customize\_Color\_Control class to create a color selector element.

1. $wp\_customize->add\_control( **new** WP\_Customize\_Color\_Control( $wp\_customize, 'background\_color', **array**(
2. 'label'        => 'Background Color',
3. 'section'    => 'cd\_colors',
4. 'settings'   => 'background\_color',
5. ) ) );

The first parameter of this object should be the **$wp\_customize** object itself, the second should be the controls id. The third array contains some options like the label, the id of the section it will be placed in and the id of the setting it controls.

### GENERATING THE CSS

To make this actually work we’ll need to apply the value of the setting to our theme. In this case, we want to make sure that the body always has the background color value as our new setting.

1. add\_action( 'wp\_head', 'cd\_customizer\_css');
2. **function** cd\_customizer\_css()
3. {
4. ?>
5. <style type="text/css">
6. body { background: #<?php echo get\_theme\_mod('background\_color', '#43C6E4'); ?>; }
7. </style>
8. <?php
9. }

Note that this code should not be placed inside the **cd\_customizer\_settings()** like we did with controls, sections and settings.

The only thing to note is how the value is retrieved. The **get\_theme\_mod()** function will retrieve the current theme’s settings; the first parameter is the name of the setting, the second is the default value.

### LIVE PREVIEWS

## Live previews allow us to use Javascript to selectively modify elements. To get started, let’s create a **customizer.js** file and make sure it is enqueued.

1. add\_action( 'customize\_preview\_init', 'cd\_customizer' );
2. **function** cd\_customizer() {
3. wp\_enqueue\_script(
4. 'cd\_customizer',
5. get\_template\_directory\_uri() . '/customizer.js',
6. **array**( 'jquery','customize-preview' ),
7. '',
8. true
9. );
10. }

Next, make sure the value of the **transport** option of the setting is set to **postMessage** in **cd\_customizer\_settings** function.

Finally, paste the following into the **customizer.js** file, within the closure.

1. ( **function**( $ ) {
3. wp.customize( 'background\_color', **function**( value ) {
4. value.bind( **function**( newval ) {
5. $( 'body' ).css( 'background-color', newval );
6. } );
7. } );
9. } )( jQuery );

The **customize()** function takes two parameters, the name of the setting to listen to, and a function that performs an action. The function in turn fires another function that binds the value of our setting and allows us to use it at our leisure.

#### LIVE PREVIEW FOR CORE FUNCTIONALITY

The ability to modify the title and blog description already exists. Let’s apply the live preview to these elements as well.

While we didn’t define these options and controls we do still have the means to modify them. We’ll use the **get\_setting()** method from the customizer class to get and modify options for a setting.

1. $wp\_customize->get\_setting( 'blogname' )->transport = 'postMessage';
2. $wp\_customize->get\_setting( 'blogdescription' )->transport = 'postMessage';

The code above should be added within the **cd\_customizer\_settings()** function. The final step is to add the Javascript code that will be fired whenever the setting’s value is changed. For example:

1. wp.customize( 'blogname', **function**( value ) {
2. value.bind( **function**( newval ) {
3. $( '#intro h1' ).html( newval );
4. } );
5. } );
7. wp.customize( 'blogdescription', **function**( value ) {
8. value.bind( **function**( newval ) {
9. $( '#intro h2' ).html( newval );
10. } );
11. } );

### SHOWING/HIDING SECTIONS

To show/hide an element we can chosen to create a radio button element. We’ll be using two parameters for the **add\_control()** which means that the first one is a simple string, not a control object. The second parameter will have all the details the control needs to build the UI. Here’s the full code which should be placed in **the cd\_customizer\_settings()** function.

1. $wp\_customize->add\_section( 'cd\_button' , array(
2. 'title'      => 'The Button',
3. 'priority'   => 20,
4. ) );
6. $wp\_customize->add\_setting( 'cd\_button\_display' , array(
7. 'default'     => **true**,
8. 'transport'   => 'refresh',
9. ) );
11. $wp\_customize->add\_control( 'cd\_button\_display', array(
12. 'label' => 'Button Display',
13. 'section' => 'cd\_button',
14. 'settings' => 'cd\_button\_display',
15. 'type' => 'radio',
16. 'choices' => array(
17. 'show' => 'Show Button',
18. 'hide' => 'Hide Button',
19. ),
20. );

In **index.php** we need to wrap the button in a conditional statement that checks the value of the setting and displays the button appropriately.

1. <?php **if**( get\_theme\_mod( 'cd\_button\_display', 'show' ) == 'show' ) : ?>
2. <a href="" **class**='button'>Come On In</a>
3. <?php endif ?>

Note that for now we’ve set the transport method to refresh. When the button is hidden it is not loaded at all on the page so we’d need to do more than just show/hide an element via JavaScript.

#### TEXT MODIFICATIONS

Continuing on with our button modifications, let’s make the text editable. We already have the button section, all we need is a setting and a simple text field control.

1. $wp\_customize->add\_setting( 'cd\_button\_text' , **array**(
2. 'default'     => 'Come On In',
3. 'transport'   => 'postMessage',
4. ) );
6. $wp\_customize->add\_control( 'cd\_button\_text', **array**(
7. 'label' => 'Button Text',
8. 'section'   => 'cd\_button',
9. 'type'   => 'text',
10. ) );

To build the option into the theme we’ll replace the hard-coded text with the **get\_theme\_mod()** function and define an appropriate default.

1. <a href="" **class**='button'><?php echo get\_theme\_mod( 'cd\_button\_text', 'Come On In' ) ?></a>

Finally, lets take care of the live preview in the **customizer.js** file.

1. wp.customize( 'cd\_button\_text', **function**( value ) {
2. value.bind( **function**( newval ) {
3. $( '#intro a' ).html( newval );
4. } );
5. } );

## SHORTCODE

This is a very helpful API it allows we to create snippets with square brackets that do stuff, basically.

They are WordPress-specific pieces of code that allow we to save time and do things that would otherwise require technical knowledge and possibly large amounts of code. By adding shortcodes to our themes or plug-ins.

### CREATING

We have to define a handler function that parses the shortcode and returns some output. We then register the shortcode using the **add\_shortcut()** function. A shortcut handler function accepts up to three arguments, listed below:

* **$atts:** An array of any shortcode attributes
* **$content:** Any content that the shortcode may enclose
* **$code:** The shortcode name (used usually when the same handler is used for multiple shortcode

For example:

1. [worldskills]
   * **$atts** = array()
   * **$content** = null
2. [worldskills]Lorem ipsum dolor sit amet[/worldskills]
   * **$atts** = array()
   * **$content** = Lorem ipsum dolor sit amet
3. [worldskills id='555' name='some name']
   * **$atts** = array('id' => '555', 'name' => 'some name')
   * **$content** = null
4. [worldskills id='555']Lorem ipsum dolor sit amet[/worldskills]
   * **$atts** = array('id' => '555')
   * **$content** = Lorem ipsum dolor sit am

Now we can go ahead and create our first shortcode.

To begin, open up the **functions.php** file of the theme, and add this line of code to the top:

1. **include** 'shortcodes.php';

To make things easier, all our shortcodes will be in this file. To begin, we will be creating a shortcode that displays the most recent blog post. The advantage of this is that every time a new post comes out, no manual updating is required. Here's the code:

1. **function** worldskills\_recentpost($atts, $content=null){
3. $getpost = get\_posts( **array**('number' => 1) );
5. $getpost = $getpost[0];
7. $return = $getpost->post\_title . "<br />" . $getpost->post\_excerpt . "...";
9. $return .= "<br /><a href='" . get\_permalink($getpost->ID) . "'><em>read more →</em></a>";
11. **return** $return;
13. }
14. add\_shortcode('newestpost', 'worldskills\_recentpost');

What we do is use the **get\_posts()** function to retrieve the most recent post and display the title, excerpt and a link to the post. The shortcode has no attributes, but we could, as an example, add attributes to display posts from a particular category or display a user-defined number of newest posts. To get the **$atts**,we use this code:

1. extract(shortcode\_atts( **array**('id' => ''), $atts));

By default, WordPress scans the post content for shortcodes and applies them. However, there are two cases in which we may want to explicitly tell WordPress to parse shortcodes in a string. For this, we need the do\_shortcode($content) function, for example:

1. **<div** class="recent"**>**
2. **<h3>**NEWEST POSTS**</h3>**
3. **<ul>**
4. **<?**= do\_shortcode('[newestpost]') **?>**
5. **</ul>**
6. **</div>**

### REMOVING SHORTCODE

Shortcodes can be removed using the **remove\_shortcode()** function. Usage is **remove\_shortcode($shortcodename)**. We can also remove all shortcodes by using the function **remove\_all\_shortcodes()**

# CUSTOMIZE THE THEME

## FEATURED IMAGE

We might want to have a featured image for each blog post. This functionality is not built into the WordPress core, but is extremely easy to implement. Place this code in our **functions.php**.

1. // Support Featured Images
2. add\_theme\_support( 'post-thumbnails' );

If we need to display the image in **content-single.php.**

1. <?php **if** ( has\_post\_thumbnail() ) {
2. the\_post\_thumbnail();
3. } ?>

## CUSTOM POST TYPES

One of the most versatile way to extend our WordPress site as a full blown content management system is with custom post types. A custom post type is the same as Posts, except we can add as many of them as we want, and with as much custom functionality as we want.

In **functions.php**, we’re going to create the custom post type called **My Custom Post**.

1. // Custom Post Type
2. **function** create\_post\_custom\_post() {
3. register\_post\_type( 'custom\_post',
4. **array**(
5. 'labels'       => **array**(
6. 'name'       => \_\_( 'Custom Post' ),
7. ),
8. 'public'       => true,
9. 'hierarchical' => true,
10. 'has\_archive'  => true,
11. 'supports'     => **array**(
12. 'title',
13. 'editor',
14. 'excerpt',
15. 'thumbnail',
16. )
17. )
18. );
19. }
20. add\_action( 'init', 'create\_post\_custom\_post' );

In the **create\_post\_custom\_post(),** we’ve created a post called **My Custom Post** with a slug of **my-custom-post**. If the original URL was example.com, the custom post type would appear at example.com/my-custom-post.

In **supports**, we can see what we’re adding – title, editor, excerpt, and thumbnail. These translate to the fields on the back end that will be available.

* Supports a **title** – the\_title(), **editor** – the\_content(), **excerpt** – the\_excerpt(), and **thumbnail**/featured image – the\_post\_thumbnail().
* Supports taxonomies, or ways to group posts, with tags and categories

Now we need to make a new page for the custom post to loop in called **Custom**, which will appear at example.com/custom.

We’re going to create **page-custom.php**, and copy the code over from **page.php**. According to the WordPress template hierarchy, a **page-name.php** will override **page.php**.

A custom post type loop will look like this:

1. <?php
3. $args = **array**(
4. 'post\_type' => 'custom\_post',
5. 'orderby' => 'menu\_order',
6. 'order' => 'ASC'
7. );
9. $custom\_loop = **new** WP\_Query( $args );
11. **if** ( $custom\_loop->have\_posts() ) : **while** ( $custom\_loop->have\_posts() ) : $custom\_loop->the\_post();
12. $meta = get\_post\_meta( $post->ID, 'custom\_fields', true ); ?>
14. <!-- contents of Our Post -->
16. <?php **endwhile**; **endif**; wp\_reset\_postdata(); ?>

## CUSTOM METABOX

### ADDING

Here’s the code to add a meta box in **functions.php** file.

1. <?php
3. **function** add\_custom\_fields\_meta\_box() {
4. add\_meta\_box(
5. 'custom\_fields\_meta\_box', // $id
6. 'Custom Fields', // $title
7. 'show\_custom\_fields\_meta\_box', // $callback
8. 'custom\_post', // $screen
9. 'normal', // $context
10. 'high' // $priority
11. );
12. }
13. add\_action( 'add\_meta\_boxes', 'add\_custom\_fields\_meta\_box' );

The actual function here is **add\_meta\_box( $id, $title, $callback, $screen, $context, $priority )**. The most important ones are $**title**, which is Custom Fields, and **$screen** (or page), which is where the meta box will be added.

### SAVE FIELDS IN THE DATABASE

First, the function that will display all our custom fields.

In **functions.php**, develop something like this:

1. <?php
3. **function** show\_custom\_fields\_meta\_box() {
4. **global** $post;
5. $meta = get\_post\_meta( $post->ID, 'custom\_fields', true ); ?>
7. <input type="hidden" name="custom\_meta\_box\_nonce" value="<?php echo wp\_create\_nonce( basename(\_\_FILE\_\_) ); ?>">
9. <!-- All fields will go here -->
11. <?php }

Now, we’re going to paste in this big chunk of code that will save all **custom\_fields** to the database.

1. <?php
3. **function** save\_custom\_fields\_meta( $post\_id ) {
4. // verify nonce
5. **if** ( !wp\_verify\_nonce( $\_POST['custom\_meta\_box\_nonce'], basename(**\_\_FILE\_\_**) ) ) {
6. **return** $post\_id;
7. }
8. // check autosave
9. **if** ( defined( 'DOING\_AUTOSAVE' ) && DOING\_AUTOSAVE ) {
10. **return** $post\_id;
11. }
12. // check permissions
13. **if** ( 'custom\_post' === $\_POST['post\_type'] ) {
14. **if** ( !current\_user\_can( 'edit\_page', $post\_id ) ) {
15. **return** $post\_id;
16. } **elseif** ( !current\_user\_can( 'edit\_post', $post\_id ) ) {
17. **return** $post\_id;
18. }
19. }
21. $old = get\_post\_meta( $post\_id, 'custom\_fields', true );
22. $new = $\_POST['custom\_fields'];
24. **if** ( $new && $new !== $old ) {
25. update\_post\_meta( $post\_id, 'custom\_fields', $new );
26. } **elseif** ( '' === $new && $old ) {
27. delete\_post\_meta( $post\_id, 'custom\_fields', $old );
28. }
29. }
30. add\_action( 'save\_post', 'save\_custom\_fields\_meta' );

Make sure **custom\_meta\_box\_nonce** matches the name attribute, and we’ve specified **custom\_fields** in the meta box function. This code is verifying the Nonce from the first function, making sure the user has the correct permissions to update the fields, and updating the post meta fields.

### CREATE CUSTOM FIELDS

Now we’re going to return to <!-- All fields will go here -->. This is where we’re going to make our **input text field, textbox, checkbox, select menu, and image upload**. Right above that, we created a variable called **$meta**. This reaches into the **custom\_fields** table in the database and retrieves the information**: $meta = get\_post\_meta( $post->ID, 'custom\_fields', true );**. We’re going to make an array and put all of our custom fields in it.

#### TEXT INPUT

We’re going to add a regular text input. The **regular-text** class is just a built in WordPress admin style. Whatever we put in the straight brackets will be the code for our custom field. If we wanted to make this text field an e-mail address, for example on a custom post called “Team Members” with contact information for each team member, we might call it custom\_fields[email] instead of custom\_fields[text].

1. **<p>**
2. **<label** for="custom\_fields[text]"**>**Input Text**</label>**
3. **<br>**
4. **<input** type="text" name="custom\_fields[text]" id="custom\_fields[text]" class="regular-text" value="<?php echo $meta['text']; ?>"**>**
5. **</p>**

#### TEXTAREA

The code for the text area is almost the same as the input, except the value is echoed out between the tags instead of as an attribute. The rows, cols, and style placed on it doesn’t really matter. It’s important to leave no space between the tags, to ensure no extra space ends up in our textbox.

1. **<p>**
2. **<label** for="custom\_fields[textarea]"**>**Textarea**</label>**
3. **<br>**
4. **<textarea** name="custom\_fields[textarea]" id="custom\_fields[textarea]" rows="5" cols="30" style="width:500px;"**><?php** echo $meta['textarea']; **?>**
5. **</textarea>**
6. **</p>**

#### CHECKBOX

There might be several ways to implement the checkbox, but this is one way that is quickly to develop.

1. **<p>**
2. **<label** for="custom\_fields[checkbox]"**>**Checkbox
3. **<input** type="checkbox" name="custom\_fields[checkbox]" value="checkbox" **<?php** if ( $meta['checkbox'] === 'checkbox' ) echo 'checked'; **?>>**
4. **</label>**
5. **</p>**

#### SELECT MENU

We can include as many options as we want here, but for now we just doing two for the example.

1. **<p>**
2. **<label** for="custom\_fields[select]"**>**Select Menu**</label>**
3. **<br>**
4. **<select** name="custom\_fields[select]" id="custom\_fields[select]"**>**
5. **<option** value="option-one" **<?php** selected( $meta['select'], 'option-one' ); **?>>**Option One**</option>**
6. **<option** value="option-two" **<?php** selected( $meta['select'], 'option-two' ); **?>>**Option Two**</option>**
7. **</select>**
8. **</p>**

### DISPLAY THE OUTPUT

Now, we have all our fields, and they all appear in the Custom Fields meta box.

These will all go in the **$custom\_loop** query. Make sure **$meta = get\_post\_meta( $post->ID, 'custom\_fields', true );** is in our loop.

Now we’re going to return to <!-- contents of our Post --> and simple output the text, textarea, checkbox, select, and an image.