Gutenberg Full Site Editing and Block-Based Themes

Note

Right now block-based themes and full site editing are still work in progress. I write about it because, sooner rather than later, themes will take advantage of full site editing and they will become a new tool in the arsenal for WordPress developers.

These features are not part of Core WordPress, they require the latest Gutenberg plugin (not the version that is bundled with WordPress) and the APIs discussed in this post may change before they are merged into core and become official parts of WordPress.

An experimental feature available in recent versions of Gutenberg is the ability to create block-based themes

The idea is that themes will leverage blocks, patterns and other Gutenberg features to enable easier development and customization of a theme.

A related feature, also under development, is Full site editing (`fse`). According to the fse page in the WordPress Design Handbook:

> The goal of the full site editing project is to utilize the power of Gutenberg's block model in an editing experience beyond post or page content. In other words, the idea is to make the entire site customizable. This editing mode will understand the structure of the site and provide ways to modify global elements like headers and footers.

In the current model, different parts of the ecosystem provide different functionality: themes provide the logical structure of a site, plugins extend the functionality of the site and the editor (classic or Gutenberg) allow you to create content for the site within the boundaries of the theme and with the functionality provided by the core WordPress and plugins.

With `fse` things change. We now can use blocks to create other types of content like headers, footers and sidebars.

With these changes, the way we create themes also changes. In this post I will talk about block-based themes and full site editing, how it works now, and what we can expect in the future

Basic Requirements

A block based theme requires an `index.php` file, an `index template file`, a `style.css` file, and a `functions.php` file.

The theme may optionally include an experimental-theme.json file to manage global styles.

The basic requirement to edit a block theme is to have a theme that is already configured to work with blocks.

For these examples, I generated an empty theme with the block theme generator. I then copied the `experimental-theme.json` styles from thett1-block theme to make sure that the styles work while I research the new syntax and whether that's absolutely necessary or I can still work with CSS on a global CSS file.

Templates and template parts

Block-based themes replace templates written in PHP and HTML with HTML files that contain HTML and Gutenberg markup.

Templates are placed inside the `block-templates` folder, and template parts go in the `block-template-parts` folder:

```
theme
|__ style.css
|__ functions.php
|__ index.php
|__ experimental-theme.json
|__ block-templates
    |__ index.html
```

```
    |__ single.html
    |__ archive.html
    |__ ...
|__ block-template-parts
    |__ header.html
    |__ footer.html
    |__ sidebar.html
```

Each template is made of the raw markup for one or more blocks and HTML.

To create the raw HTML blocks you need to either learn their syntax or copy them from the editor.

In the following example, the comments containing `wp:` indicate the opening and closing instruction for a Gutenberg block.

The following example has three elements

- A `columns` container that will hold the child column elements and their content
- Two `column` (singular) elements for each content and its column
    - The column on the left has an embed element for Youtube content
    - The column on the right has a paragraph

```
<!-- wp:columns -->
<div class="wp-bloc<div class="wp-block-columns"><!-- wp:column -->
<div class="wp-block-column">
<!-- wp:embed {"providerNameSlug":"youtube","responsive":true} /--></div>
<!-- /wp:column -->

<!-- wp:column -->lock-column"><!-- wp:paragraph -->
<p>The idea is to have text and a video</p>
<!-- /wp:paragrap<!-- wp:paragraph --><!-- /wp:paragraph --></div>
<!-- /wp:column --></div>
<!-- /wp:columns -->
```

This part addresses the structure, not the style. We'll discuss styling later, when we talk about `experimenta-theme.json` styles and CSS styles.

(experimental-)theme.json

The experimental-theme.json presents a unified (although harder to understand) way to present styles for a block-based theme.

According to the documentation:

> The Block Editor API has evolved at different velocities and there are some growing pains, specially in areas that affect themes. Examples of this are: the ability to control the editor programmatically, or a block style system that facilitates user, theme, and core style preferences.
>
> This describes the current efforts to consolidate the various APIs related to styles into a single point – a experimental-theme.json file that should be located inside the root of the theme directory.

```
{
  "templateParts": {
    "header": {
      "area": "header"
    },
    "header""footer"  "area": "foote": "   }
  },
  "settings": "settings""settings": {
    "defaults": {
      "color": {
        "palette"      "slug": "black",
     ": ""slug"lor": "#000000",
      ": ""color"": "Black"
          },
 ": ""name": "Black"
          },
          // More colors defined here
        ],
        "gradients": "purple-to-yellow",
      "purple-to-yellow""slug": "purple-to-yellow",
```

```
          "gradient"purple), var(--wp--preset--color--yellow))",
          "name": "Purple to Yellow"
        },
        //": ""name": "Purple to Yellow"
        },
        // More gradients defined here
      ]
    },
    "typography": {
      "customLineHeight"slug": "extra-sm": ""fontSizes"   "size": "16px"
       ": ""slug" "Extra small"
        },
"Extra small""size": "16px",
          "name"   ],
      "fontFamilies": [
        {
          "fontFamily": "-apple-system,BlinkM": ""fontFamilies"oe UI\",B
          "slug": "system-font",
          "name": "System Font"
        },
        /",Roboto,Oxygen-Sans,Ubuntu,Cantarell,\"Helvetica Neue\",sans-s
 "customPadding""name": {
      "font-primary": "-apple-system, BlinkMacSystemFont, 'Segoe UI', Ro
      "customPadding": true
    },
    "custom": {
      "font-primary"
        "heading": 1.3,
        "page-title": 1.1
      },
      "responsive": {
        "aligndefault-width": "610px"": ""line-height"nwide-width": "124
        "heading"": {
        "unit": "20px"": ""unit"  "horizontal": "25px",
": ""responsive"al": "30px"
 ": ""aligndefault-width": "610px",
        "alignwide-width"color": {
      "background": "var(--": ": {
```

```
          "unit")",
        "text": "var(-": ""horizontal": "25px",
          "vertical"": "var(--wp--preset--color--dark-gray)"": ""styles":
    "root": {
      "color"ntSize": "va": ""fontSize""background"-normal)",
        "lineHeight": "var(--wp--c": ""text": "var(--wp--preset--color--da
        "link"{
      "typography": {
        "fontSize": "var(--wp--": ""typography": {
        "fontSize" "lineHeight": "var(--wp--custom--line-height--pag": ""
      }
    },
    "core/heading/h1": {
      "typography" "title": "P": ""fontSize": "var(--wp--preset--font-size
        "lineHeight": "var(--wp--custom--line-height--page-title)"
      }
    },
    // More block styles defined here
  },
  "customTemplates": {
    "page-home": {
      "title": "Page without title"
    }
  }
}
```

Both settings and styles can contain subsections for any registered block. As a general rule, the names of these subsections will be the namespaced block names or “block selectors”.

For example, the paragraph block or can be addressed in the settings using the block selector `core/paragraph`

```
{
  "styles": {
    "core/paragraph": {}
  }
```

```
}
```

There are a few cases like the heading block where the block represents h1 to h6 HTML elements. In these cases, the block will have as many block selectors as different markup variations — `core/heading/h1`, `core/heading/h2`, etc, so they can be addressed separately.

```
{
  "styles": {
    "core/heading/h1": {},
    "core/heading/h6": {},
  }
}
```

The question about these blocks defining types of an element is whether you can define common properties and then define item specific ones, like this block of CSS:

```
/* We define the font-family for all the fonts */
h1, h2, h3, h4, h5, h6 {
  font-family: "Recursive", Verdana, sans-serif
}

"Recursive"h1 {
  font-size: 3em;
}

h2 {
  font-size: 2.5em
}
```

Additionally, there are two other block selectors: `root` and `defaults`.

- The `root` block selector represents the root of the site.
- The `defaults` block selector represents the defaults to be used by blocks if they don't declare anything.

One final note. As far as I can tell, the values in `experimental-theme.json` will override values in CSS or defined in `functions.php`.

(experimental-)theme.json: Settings

We'll cover the settings and styles separately. We will not go into details about individual settings, if you want to dive into them, the theme.json documentation goes into a lot more detail about the invididual settings.

This example contains all the settings with their default values. The only change I made was to move the custom blocks of the defaults to the top; I will explain why later.

```
{
  "settings": {
    "defaults": {
      "custom": {},
      "layout": { /* Default layout to be used in the post editor */
        "contentSize": "800px",
        "800px""wideSize": "1000px",
      }
      "border": {
        "customRadius": false /* true to opt-in */
      },
      "color": {
        "custom": true, /* false to opt-out, as in add_theme_support('disa
        "customGradient": true, /* false to opt-out, as in add_theme_suppo
        "gradients": [], /* gradient presets, as in add_theme_support('edi
        "link": false, /* true to opt-in, as in add_theme_support('experin
        "palette": [], /* color presets, as in add_theme_support('editor-c
      },
      "spacing": {
        "customPadding", /* true to opt-in, as in add_theme_support('custo
        "units": [ "px": [ "", "rem", "vh", "vw" ], /* filter values, as t
      },
      "typography":", ""typography""typography": {
        "customFontSize": true, /* false to opt-out, as in add_theme_suppo
        "customFontWeight": true, /* false to opt-out */
        "customFontStyle": true, /* false to opt-out */
```

```
        "customLineHeight": false, /* true to opt-in, as in add_theme_supp
        "dropCap": true, /* false to opt-out */
        "fontFamilies": [], /* font family presets */
        "fontSizes": [], /* font size presets, as in add_theme_support('e
      }
    }
  }
}
```

The custom section of the default settings allows you to create custom properties for the values inside the section.

For example the following custom settings in the default settings:

```
{
  "settings": {
    "defaults": {
      "custom": {
        "base-font": 16,
        "line-height": {
          "small": 1.2,
          "medium": 1.4,
          "large": 1.8
        }
      }
    }
  }
}
```

Will produce the following custom properites in the CSS `:root` element.

Note the structure of the custom properties:

- the `--wp` prefix
- the path to the property separated by `--`
- The value of the property

```
:root {
  --wp--custom--base-font: 16;
  --wp--custom--line-height--small: 1.2;
  --wp--custom--line-height--medium: 1.4;
  --wp--custom--line-height--large: 1.8;
}
```

The second thing I want to point out is that these defaults can also be added to individual blocks by adding defaults to the corresponding block selector. This gives us a finer level of control over the settings for our blocks and themes.

Styles: CSS all the things

Having the `experimental-theme.json` provides a centralized way to style a theme but it's not the only way and may not be the best way.

You can use CSS to style blocks and themes as a two step process. The first one is to add PHP with the palette definition to your theme's `functions.php`.

This will show the colors in the editor while you're adding content.

```
<?php
add_theme_support( 'edit'editor-color-palette'ay(
  array(
    'name' => esc_attr__( 'strong magenta', 'themeLangDo'name'),
    'slug' => 'strong-magenta',
    'color' => ''slug'#a156b4',  array(
    'name' => esc_attr__( 'light grayish magenta', 'themeLangDomain' ),
    'slug' => 'light-grayish-magenta',
    'color' => '#d0a5d'name'#d0a5db',y(
    'name' => esc_attr__( 'very light gray', 'themeLangDomain' ),
    'slug' => 'very-light-gray',
    'color' => '#eee',
  ),
'name'#eee', 'name' => esc_attr__( 'very dark gray', 'themeLangDomain' ),
    'slug' => 'very-dark-gray',
    'color' => '#444',
```

```
  ),
) );
'name'#444',
  ),
) );
```

The next step is to add CSS that will handle each color as a background and text color.

```
.has-strong-magenta-background-color {
    background-color: #a156b4;
}

.has-strong-magenta-color {
    color: #a156b4;
}
```

The documentation for theme_support outlines other areas where we can use this combination to style blocks and block content.

Styles: which one to use?

When working with block-based themes it appears that the `experimental-theme.json` way is the way to go but it is not intuitive and there is no good documentation for it or, at least, no documentation that would make sense to beginners.

I will continue to research alternatives but, for now, the `theme.json` file seems to be the best alternative regarding theme styles for a block-based theme.

And since `theme.json` will supersede other styles, it's worth diving deeper into the format and some outstanding questions:

- How do you incorporate local custom fonts?
- How do you incorporate variable font definitions into the theme.json syntax?

Creating blocks with an external declaration

The block type metadata provides an external means to declare our block API that will also be necessary when (and if) you decide to submit it to the block directory.

The metadata is stored in a `block.json` file. An example, taken from the Block metadata developer docs looks like this:

```
{
  "apiVersion": 2,
  "name": "my-plugin/notice",
  "my-plugin/notice""title"egory": "text"": ""category"[ "core/group" ],
  "core/group"r",
  "description": "": ""description""icon": "star",
  "description"ywords": [ "alert", "message" ],
  "rivendellweb": [ ""keywords": [ "alert", "message" ],
  "rivendellweb-blocks"type": "string",
": ""attributes""html",
"html""message"": ".messa": ""type": "string",
      "source": "html",
      "selector" "message"
  },
  "usesCon"message""providesContext": {
    "my-plugin/message": "message"
  },
  "usesContext"e": "default", "lab": ""supports": {
    "align": true
  },
  "styles""other", ""other""name": "default", "label": "Default", "isDefau
      "message": "Thi": ": "other", "label"  },
  "editorScript": "file": ""editorScript": {
    "attributes": {
      "message"
  "editorStyle": "file:./build/ind": ""editorScript": "file:./build/style
}
"file:./build/style.css""script": "file:./build/script.js",
```

```
  "editorStyle": "file:./build/index.css",
  "style": "file:./build/style.css"
}
```

We then register the block

```
<?php
register_block_type_from_metadata(
    __DIR__ . '/notice',
    array(
        'render_callback' => 'render_block_core_notice',
    )
);
```

Look at the Block metadata developer docs for more information about the content of the file and how it works.

I've moved a section about block development and integration via metadata JSON files to a separate post as it needs more reseach before deciding if I want to use it or not but there are some thoughts that go hand-in-hand with creating a theme: Do we create plugins for individual blocks or create block libraries for a single theme.

Single block plugins or block libraries?

When writing blocks we can take one of two paths:

- We use the single-responsibility principle to write single-purpose blocks that perform a single tasks.
- Build larger library plugins that contain multiple blocks and related functionality in a single package.

Each of these options presents tradeoffs.

If we do a plugin for each block, we get smaller plugins but we get more of them... if they are all loaded in sequence we might get some slowdown because they each have to be loaded by the browser, parsed by WordPress and loaded by WordPress.

If we choose a library plugin approach for the blocks you design you subject yourself to many more updates and having to update the plugin if any of the components bundled in it change.

When working on a plugin library I treat each block as its own NPM package (since it uses Javascript in addition to PHP) and manage them with Lerna to build all children packages together or jump into a single repository to build only that package.

As with many of these choices, what you need to do will depend on the needs for your project. I tend to work on libraries since I build collections of blocks neither as testing environments or for production.

Third party libraries, in-house development or a mix of both?

Blocks will soon become the new revenue generator for WordPress developers. You can search the WordPress plugin repository for blocks although I recommend you check the results to make sure they are Gutenberg blocks and not blocks for other tools.

But as a theme developer you have to ask yourself what would work better, have all your blocks developed in-house or if we should bring in blocks like these into our themes and development environments:

- Genesis Blocks from Studio Press, the same people who created the Genesis Framework
- CoBlocks by GoDaddy

For me this is a two-pronged issue. On the one hand I don't want to reinvent the wheel but on the other hand I like to have control over what goes on my themes and, even though I have access to the source code for these plugins, I'm not confident that extracting the code for specific functionality on these plugins will work without any technical and non-technical problems later on.

There is another type of block generator that presents a different set of issues. The Advanced Custom Fields (ACF) Blocks will only work if you have the ACF Pro plugin enabled on your system. This is a paid upgrade to the traditional ACF plugin and I'm not totally sure that the blocks would work in a theme without the ACF Pro plugin installed.