Revisiting Gutenberg blocks part 1: Building and Styling the blocks

My dislike of Gutenberg has softened since it was first introduced, mostly because they provided ways to keep the classic editor and they've guaranteed support for it through 2022.

I've written about building Gutenberg blocks: Building Gutenberg Blocks (Part 1), part 2, part 3 and part4, and another series on using Gutenberg as a design syste: Gutenberg as a design system (part 1), part 2, and part 3 but I want to revisit block design both in light of new developments in Gutenberg and as a way to build more complex blocks using block patterns and, possibly, full page designs.

Build as a plugin or bundle in a theme?

As with any customizations to WordPress the first question is how to bundle and package it. WordPress presents developers with two options:

- Blocks can added to a theme, making the blocks an exclusive part of a single theme and forcing you to stay in that theme or losing the ability to edit existing posts using those blocks.
- The other option is to put all block-related content in a plugin. This has the advantage of being theme independence and allowing for easier customization.

Which option you choose depends on the type of project you're working on. If I'm creating a theme for a business and don't expect to change it for a while, then using a theme to hold all your changes together makes more sense.

On the other hand, if I'm creating reusable components for myself or third parties, or if I want to add the blocks to existing libraries then a plugin makes more sense.

In the end, it's up to you :)

Revisiting block development

Rather than pointing you to different blocks over and over, I thought I'd review my process for block development here and updated with new things I've learned over since those posts were published.

The PHP side of the equation

Even though the code is written mostly in Javascript, we still need some PHP so the React/Javascript code can talk to the WordPress backend and make everything work together.

The root PHP file where we load all the blocks

The first PHP file is the root index.php that tells WordPress this is a plugin. The top of the file has the plugin metadata comment in it.

```
<?php
/**
 * Plugin Name: Rivendellweb Blocks
 * Plugin URI: https://example.com
 * Description: Sample blocks collection.
 * Version: 0.0.1
 * Author: Carlos Araya
 *
 * @package rivendellweb-blocks
 *
?>
```

Once you've added the metadata you can activate the plugin in the backend. It won't do anything, but at least we know it works.

The next step is to prevent direct access to the plugin. If the constant ABSPATH is not present, meaning that the file was accessed from outside WordPress, we exit.

```
<?php
```

```
if ( ! defined( 'ABSP'ABSPATH'{
  exit;
}
?>
```

The next section is eclusive to Gutenberg. The editor offers you categories to organize the available blocks.

For these examples I want to ccreate a custom category for my custom blocks; I also want to use these blocks only on posts so if the post type is not `post` we just return the existing categories

Otherwise, we merge the existing categories with our custom array of array of posts types. In this case I created a single category but I could add more categories for the blocks I'll create later.

The last step is to use the `block_categories` filter to apply the function that adds the custom category.

```
<?php
function rivendellweb_blocks_block_category( $categories, $post ) {
  if ( $post->post_type !== 'post'post'      return $categories;
  }
  return array_merge(
    $categories,
    array(
      array(
        'slug' => 'rivendellweb-blocks',
        'title' => __( 'Rivendellweb Blocks', 'rivendellweb-blocks' ),
        ''slug' => 'wordpress',
      ),
    )
  );
}
add_filter( 'block_categories', 'rivendellweb_blocks_block_category', 10,
'wordpress'?>
```

The final step is to include the blocks we create. This example has seven blocks and

we include the index file for each.

I evaluated if I should use include_once rather than just include but the files are loaded only once so it's not needed.

```
<?php
include 'exam'example-01/index.php'lude 'example-02/index.php';
include 'example-'example-02/index.php' 'example-04/index.php';
include 'example-05/index.php';
include 'example-06/index.php';
include 'example-07/i'example-04/index.php'?>
```

Whenever I add a new block to the plugin I need to include the block's `index.php` file.

The PHP code for each blocks

This is the example of a working block with translations to Spanish.

Once again, I don't want users or possibly bad actors to access the block directly so if the ABSPATH constant is not set then the code exits early and is done.

```
<?php
if ( ! defined( 'ABSPATH' ) ) {
    exit;
}
```

The next step is to load the translation files for the plugin using load_plugin_textdomain().

```
function rivendellweb_blocks_example_01_load_textdomain() {
    load_plugin_textdomain( 'rivendellweb-blocks', false, basename( __DIR_
}
```

The next function does most of the work.

We first check if the register_block_type() function exists. If it doesn't we return early, there's nothing for the script to do.

Next, we assing the assets file to a variable that will be used when we register the block. The block's build process will generate the assets file.

The script calls wp_register_script() to register the script with WordPress without loading it.

register_block_type will add the block javascript and CSS behind the scenes.

If the wp_set_script_translations function exists then it sets the translated scripts for the block.

The final step is to run the function to WordPress' init hook to make sure that WordPress has loaded before the block is added and loaded.

```
<?php
function rivendellweb_blocks_example_01_register_block() {

  if ( ! function_exists( 'regi'register_block_type'{
    // Gutenberg is not active.
    return;
  }

  // automatically load dependencies and version
  $asset_file = include( plugin_dir_path( __FILE__ ) . 'build/index.asset

  wp_register_script(
      'rivendellweb-blocks-example-01',
      plugins_url( 'build/index.js', __FILE__ ),
      $asset_file['dependencies'],
      $asset_file['version']
  );

  registe'build/index.asset.php'lweb-blocks/example-01', array(
      'editor_script' => 'rivendellweb-blocks-example-01',
  ) );

  if ( function_exists('wp_set_script_translations' ) ) {
    wp_set_script_translations( 'rivendellweb-blocks-example-01', 'rivende
  }
```

```
}
add_action( 'init', 'rivendellweb_blocks_example_01_register_block' );
```

The Javascript portion of the block

Once the PHP parts of the block plugin are done we can start building the Javascript portion of the block.

The block uses ES2015 syntax and ES2015 module import syntax.

Rather than install several packages and have to configure them individually I chose to use @wordpress/scripts, a WordPress-specific version of scripts needed to work with blocks. It makes the following tools available that we can incorporate into an existing `package.json`:

```
{
  "scripts": {
    "build": "wp-scripts build",
    "check-engines": "wp-scripts check-engines",
    "check-licenses": "wp-scripts check-licenses",
    "format:js": "wp-scripts format-js",
    "lint:css": "wp-scripts lint-style",
    "lint:js": "wp-scripts lint-js",
    "lint:md:docs": "wp-scripts lint-md-docs",
    "lint:md:js": "wp-scripts lint-md-js",
    "lint:pkg-json": "wp-scripts lint-pkg-json",
    "packages-update": "wp-scripts packages-update",
    "start": "wp-scripts start",
    "test:e2e": "wp-scripts test-e2e",
    "test:unit": "wp-scripts test-unit-js"
  }
}
```

The advantage of using these scripts is that they are preconfigured with the WordPress team defaults.

Imports and style definitions

The first section of the block imports modules and defines the styles that will use the block.

__ is the Javascript version of the PHP internationalization libraries. There are examples of how it works later in the block.

This is a basic example. In more advanced blocks styles would be pulled into an external file and linked from the script.

```
import { __ } from '@wordpress/i18n';
import { registerBlockType } from '@wordpress/blocks';

const blockStyle = {
  backgroundColor: '#639',
  color: '#fff',
  padding: '20px',
};
```

Block metadata

There are two types of block metadata. The first one is what I call metadata and goes inside the block.

```
  title: __( 'Example 01', 'rivendellweb-blocks' ),
  icon: 'universal-access-alt',
  category: 'rivendellweb-blocks',
  example: {},
```

In this example there is no problem with adding the metadata directly in the block but in more advanced blocks or block patterns there may be duplication between PHP and Javascript code defining the same items so we need to have a better way to do it; later in the post, I'll address a way to create an external metadata document that we can use both in PHP and Javascript.

Editor support and block attributes

One thing that, I believe, is new in Gutenberg is the ability to customize what editor features the block supports

The `supports` example does the following:

- Allows adding IDs for custom anchor links to the element
- Adds support for all the options: left, center, right, wide, and full
- Creates the space to add a custom class name for the block

```
supports: {
  anchor: true,
  align: true,
  customClassName: true,
}
```

The idea behind this is that, when you give the blocks to other people to work with, you can tailor the editing experience for the authors.

Nested blocks

One of the things that bothered me is that, years after I filed the issue, there's still not an easy way to add lists.

I think that nested blocks may be the solution to this problem at the cost of creating custom blocks that replace default ones.

Blocks using nested/inner blocks can also be configured to only accept a subset of available blocks.

The nested blocks example allows you to register a block with images and paragraphs as child elements.

Using nested templates gives us "selective" flexibility. We can choose what elements content creators can use in specific blocks, giving them both flexibility and a constrained design environment.

```
import { registerBlockType } from '@wordpress/blocks';
import { InnerBlocks, useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'rive'@wordpress/block-editor'1', {
  const ALLOWED_BLOCKS = [
    'core/image',
    'core/paragraph'
  ];

  'core/image'// Metadata

  edit: () => {
    const blockProps = useBlockProps();
    return (
      <div { ...blockProps }>
        <InnerBlocks
          allowedBlocks={ ALLOWED_BLOCKS }
        />
      </div>
    );
  },

  save: () => {
    const blockProps = useBlockProps.save();
      return (
        <div { ...blockProps }>
          <InnerBlocks.Content />
        </div>
      );
  },
} );
```

Another thing that we can do is to use a template. This is a set of allowed blocks that are preconfigured with default placeholders that can be updated when editing the block.

Unlike `allowed_blocks`, using `block templates` gives us the same type of flexibility with the added Advantage that we can guide authors with placeholder text. We can also use the templateLock attribute to control whether authors can

change the block. The possible values for the attribute are:

- `all` makes it impossible to insert new blocks, move or delete existing blocks
- `insert` prevents adding or deleting child blocks, but allows moving existing ones
- `false` prevents locking altogether

```
import { registerBlockType } from '@wordpress/blocks';
import { InnerBlocks, useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'rive'@wordpress/block-editor'1', {
  const MY_TEMPLATE = [
    [ 'core/image', {} ],
    [ 'core/heading', {
      placeholder: 'Book Title'
    } ],
    [ 'core/paragraph', {
      placeholder: 'Summary'
    } ],
  ];
 'core/image'// Metadata

  edit: () => {
    const blockProps = useBlockProps();
    return (
      <div { ...blockProps }>
        <InnerBlocks
          template={ MY_TEMPLATE }
          templateLock="all"
        />
      </div>
    );
  },
});
```

Styling the component

One of the things that I find the most infuriating about Gutenberg blocks is that

you need two styles that will likely do the same thing.

As far as styles there are a few things that we need to do.

The first thing to do is to add editor style support for the existing theme using add_theme_support for editor styles using editor_styles and then enqueue the editor style using add_editor_style

`add_editor_style` will add the editor styles to the list of stylesheets that WordPress will load.

```
<?php
function rivendellweb_gutenberg_css(){
  // if you don't add this line, your stylesheet won't be added
  add_theme_support( 'editor-styles' );
  'editor-styles'// tries to include style-editor.css directly from your t
  add_editor_style( 'style-editor.css' );
}

add_action( 'after_setup_theme', 'rivendellweb_gutenberg_css' );
```

The next step is to decide if we need/want to remove styles from core elements and if we need/want to add styles to these core elements.

**Rather than creating custom styles. You may want to remove existing styles and set your own or add your own styles using block style variations

`unregister_block_style` allows unregistering a block style previously registered on the server using register_block_style.

The function unregister_block_style only unregisters styles that were registered on the server using register_block_style. The function does not unregister a style registered using client-side code.

The function's first argument is the registered name of the block, and the name of the style as the second argument.

The following code sample unregisters the style named 'fancy-quote' from the quote block:

```php
<?php
unregister_block_style( 'core/quote', 'fancy-quote' );
```

The opposite function, used to register styles for Gutenberg blocks is registerBlockType

THe script also uses the domReady package to ensure that this code will run after the DOM has finished loading.

```
wp.domReady( () => {
  wp.blocks.unregisterBlockStyle(
    'core/button',
    [
      'default',
      'outline',
      'squared'
    ]
  );

  wp.blocks.registerBlockStyle(
    'core/button',
    [
      {
        name: 'default',
        label: 'Default',
        isDefault: true,
      },
      {
        name: 'full',
        label: 'Full Width',
      }
    ]
  );

  wp.blocks.unregisterBlockStyle(
    'core/separator',
    [
```

```
      'default',
      'wide',
      'dots'
    ],
  );

  wp.blocks.unregisterBlockStyle(
    'core/quote',
    [
      'default',
      'large'
    ]
  );

} );
```

We will name the script `editor.js` and will use the name to enqueue it from the PHP side.

The `rivendellweb_gutenberg_scripts` function enqueues the `editor.js` script we created in the previous step and makes it depend on `wp-blocks` and `wp-dom`.

The function is hooked to the enqueue_block_editor_assets action. This will make the additions and removals in the script active and will add and remove styles from the editor.

```
<?php
function rivendellweb_gutenberg_scripts() {
  wp_enqueue_script(
    'rivendellweb-editor',
    get_stylesheet_directory_uri() . '/assets/js/editor.js',
    array( 'wp-blocks', 'wp-dom' ),
    filemtime( get_stylesheet_directory() . '/assets/js/editor.js' ),
    true
  );
}
```

```
add_action( 'enqueue_block_editor_assets', 'rivendellweb_gutenberg_scripts
```

Now that I've configured WordPress to work with the block's styles, I can look at building the actual styles.

Building the stylesheets

The main reason why I've listed editor and end-user styles separately is that it may take a while to figure out how to match the blocks as they appear in the editor with the block as they appear in the end-user interface.

Editor and end-user styles

One thing that authors need to keep in mind is that the styles in the editor may or may not correspond to the styles used when the block is published.

Gutenberg will use the default notation of `wp-block-` plus the name of the block. So a pullquote block would have the `wp-block-pullquote` class.

Your block may get additional classes identifying different aspects of the block. `is-{stylename}` shows the currently applied style for the block so we can use that naming scheme in the block's styles.

in addition, you can use block style variations to further customize the styles for blocks, both default and custom.

See A White Pixel's How to Add Custom Block Styles to WordPress Gutenberg Blocks

I've purposefully left out the choice of whether we should use SCSS/SASS, PostCSS or any other CSS preprocessors. That's a personal choice that may change the way we configure WebPack for each individual project. There may be situations where plain CSS would be enough.