Gutenberg as a design system (part 1)

One of the things I've found the most intriguing about Gutenberg is the ability to create variations of a given element and build the basic components of a design systems.

From there we can also build what Brad Frost calls molecules, compositions of one or more elements that are used together.

This post will explore how to create variations of existing blocks and how to create composite blocks for our end users.

The examples below will only work when Gutenberg is in visual mode and will not work in classic editor or when Gutenber is in HTML mode.

Block Variations

Variations are different styles or variations for a given component. The idea is that we don't need to create custom blocks and then style them, we can use CSS to change the appearance of default blocks to suit our needs.

Client Side

The quickkest way to create block variations is to use client-side techniques and let the browser do all the work.

This takes the following steps:

1. Define the block variations in the Javascript file we enqueued, `block-variations.js`
    1. If necessary we remove existing custom styles from the elements we're working on
2. Enqueue the script in `index.php`
3. Create the variations in `block-variations.js`
4. Enqueue `block-variations.css`
    1. Add styles for the variations in `block-variations.css`

Enqueueing the script uses the `enqueue_block_editor_assets` action and a callback function as shown below.

Note that the enqueue script also list dependencies that must be loaded before the script we're enqueueing. This makes sure we have no errors down the line.

```
<?php
function rivendellweb_enqueue_variatons() {
  wp_enqueue_script(
    'myguten-script',
    plugins_url( '/lib/block-variations.js', __FILE__ ),
    array( 'wp-blocks', 'wp-dom-ready', 'wp-edit-post' ),
    filemtime( plugin_dir_path( __FILE__ ) . '/lib/block-variations.js' )
  );
}
add_action( 'enqueue_block_editor_assets', 'rivendellweb_enqueue_variatons
```

The variations script, `block-variations.js` has two parts. The first one removes any prior definitions of our blocks so that our definition will be the one that works.

```
// Unregister existing styles
wp.domReady( function() {
  wp.blocks.unregisterBlockStyle( 'core/quote', 'large' );
  wp.blocks.unregisterBlockStyle( 'core/quote', 'fancy' );
} );
```

The second part registers variations for different elements. In this example we've done one for paragraphs and three for blockquotes.

`wp.blocks` is the ES5 version of this code. I chose it because using it I don't have to use WebPack with it.

Each delcaration uses `registerBlockStyle` and has two parameters:

1. The namespace and name of the block we're customizing
2. An object with the properties for the variation. The attributes we've used are
    1. **name**: the short name for the variation

2. **label**: the user visible name

```
// Paragraph variations
wp.blocks.registerBlockStyle( 'core/paragraph', {
  n'core/paragraph'raph',
  label: 'First Paragraph',
} );

'First Paragraph'// Blockquote variations
wp.blocks.registerBlockStyle( 'core/quote', {
  name: 'fancy-quote',
  label: 'Fancy Quote',
} );

wp.blocks.registerBlockStyle( 'core/quote', {
  name: 'top-bottom-quote',
  label: 'Top and Bottom',
})

wp.blocks.registerBlockStyle( 'core/quote', {
  name: 'red-quote',
  label: 'Red Quote',
})
```

Next, add the styles for each of the variations, we wrap the styles on a `:root` rule to avoid some specificity problems. Even then there are conflicts with custom font sizes, so I'm either having to use !important (ugh), figure out a way to increase specificity or let the cascade do its thing.

```
:root {
  .is-style-lede-paragraph {
    font-size: 125%;
  }

  .is-style-fancy-quote {
    border-left: 4px solid red;
  }
```

```
  .is-style-top-bottom-quote {
    border-top: 4px solid black;
    border-bottom: 4px solid black;
    border-left: none;
  }

  .is-style-red-quote {
    color: red;
  }
}
```

the final step is to enqueue the stylesheet to use in the front-end of the site. Because this is not for the editor, we need a different action to enqueue the styles.

```
<?php
function rivendellweb_variation_styles() {
    wp_enqueue_style( 'myguten-style', plugins_url( 'style.css', __FILE__
}
add_action( 'enqueue_block_assets', 'rivendellweb_variation_styles' );
```

Server Side

We can also create all the variations on the server and add the styles from an external style sheet.

```
<?php
wp_register_style(
  'variations-style',
  get_template_directory_uri() . '/variations/variation-styles.css' );
```

```
<?php
register_block_style(
    register_block_style(
      'core'core/paragraph'   array(
        'name' => 'lede-paragraph',
```

```
        'label' => 'First Paragraph',
        'style_'name'' => 'variations-style',
      )
    ),

    ' => '// Blockquote variations
  register_block_style(
    'core/quote',
    array(
      'name' => 'fancy-quote',
      'label' => 'Fancy Quote',
      'style_handle' => 'variations-style',
    )
  ),
  register_block_style(
    'core/quote',
    array(
      'name' => 'top-bottom-quote',
      'label' => 'Top and Bottom',
      'style_handle' => 'variations-style',
    )
  ),
  register_block_style(
    'core/quote',
    array(
      'name' => 'red-quote',
      'label' => 'Red Quote',
      'style_handle' => 'variations-style',
    )
  )
);
```

These types of customizations give us a full-powered design system on top of whatever custom blocks we create. In the next post we'll discuss templating using blocks.

Gutenberg as a design system (part 2)

In part 1 we discussed how to build variations for specific components.

We'll now see how we can create templates ready for the user to fill and either use as-is or modify if when needed.

Block Templates

The first type of template is at the block level. This allows the creation of complex blocks without having to do much on the PHP side.

We first import our tools like normal, the i18n utilities, registerBlockType and InnerBlocks

```
import { __ } from '@wordpress/i18n';
import { registerBlockType } from '@wordpress/blocks';
import { InnerBlocks } from '@wordpress/block-editor';
```

We then define our template in a constant. This is the structure of the block that we want to have.

We can use attributes of the blocks in the templace like the placeholder attribute that we use in the example to tell the user what we expect her to enter in each block.

```
const BLOCK_TEMPLATE = [
  [ 'core/image', {} ],
  [ 'core/heading', { placeholder: 'Book Title' } ],
  [ 'core/heading', { placeholder: 'Book Author' } ],
  [ 'core/paragraph', { placeholder: 'Summary' } ],
];
```

The block registration is no different than a custom block. The only difference is that our `InnerBlocks` declaration uses a template attribute with our

BLOCK_TEMPLATE definition as the parameter.

In this example we've chosen to lock the template so the user cannot move the inner blocks nor they can add any new block to the template.

The save methods remains the same.

```
registerBlockType( 'rivendellweb-blocks/example-06', {
  title: __('Example 06', 'rivendellweb-blocks'),
    category: 'rivendellweb-blocks',
    icon: 'translation',

  edit: ( { className } ) => {
    return (
      <div className={ className }>
        <InnerBlocks
        template={ BLOCK_TEMPLATE }
        templateLock="all" />
      </div>
    );
  },

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

Thing To Watch Out For

Out of the box, the styles in the editor will not necessarily match the styles when published. You need to be careful and make sure that the editor styles match the front-end styles or are close enough that you won't be surprised by what the

published content looks like.

Thing To Watch Out For

Be careful when following Gutenberg examples out in the wild as they are very inconsistent. Some will use ES5 rather than ES6+ and some will juse JSX while others will use React's create eleent syntax. I'm not saying either version is bad, I'm just warning readers that this the case so they are prepared, particularly if, like me, they are not fluent in React and how it works.

Looking forward: Block Patterns and Block Pattern Libraries

Block Patterns API is an API currently under development that would make custom blocks and variations available to users in similar ways to what other sites like Wix, SquareSpace, and Weebly and even page builders in WordPress like Elementor already do.

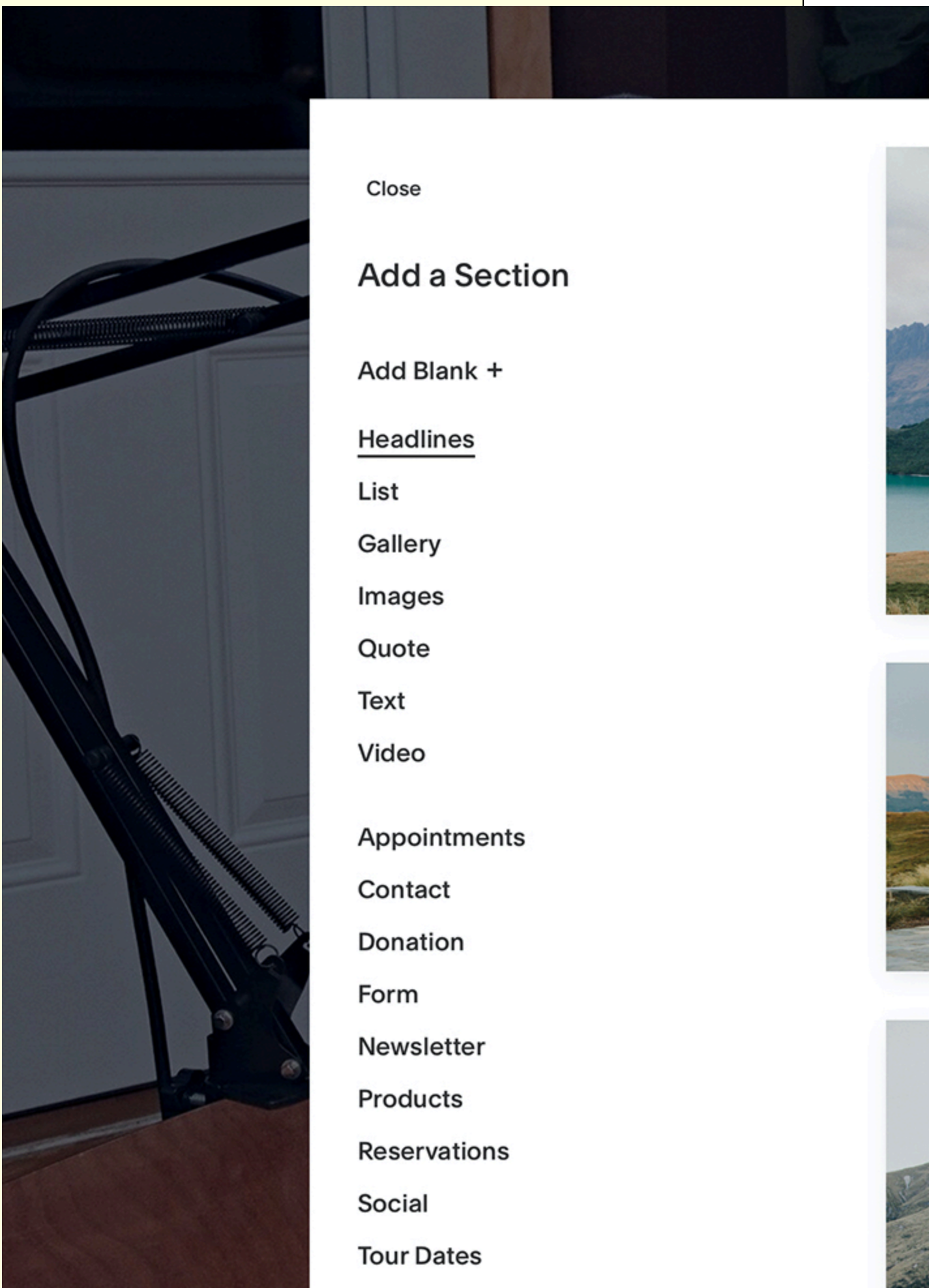

Close
Add a Section
Add Blank +
Headlines
List
Gallery
Images
Quote
Text
Video
Appointments
Contact
Donation
Form
Newsletter
Products
Reservations
Social
Tour Dates

Figure 1: Sections as they apper in SquareSpace (from Gutenberg Block Patterns: The Future of Page Building in WordPress)

Variations already give us a way to create custom styles for our existing blocks so the next logical step, to me, is to package the variations and present them to the user in a way that is more coherent to them.

Figure 2 shows a mockup of how the patterns may be surfaced to the user.

Search for a block or block pattern

Blocks | **Block Patterns**

Most used

About

Calendars and bookings

Donations

Forms

Galleries

Headlines

Lists

Newsletters

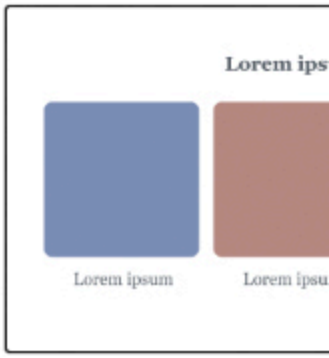
Products

Services

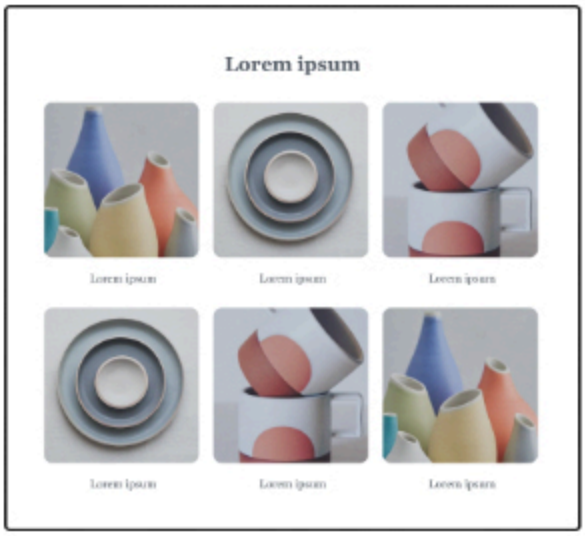
Social

Most used

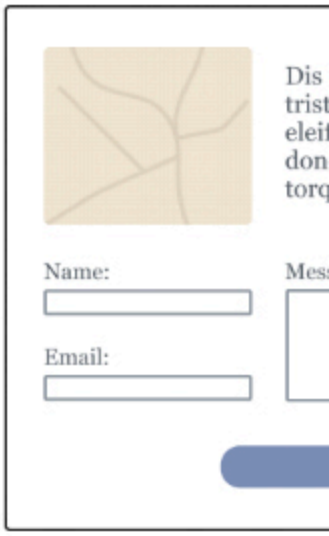
Cover



Services



Products



Contact

Figure 2: Mockup of patterns within the Gutenberg block library (from Gutenberg Block Patterns: The Future of Page Building in WordPress)

Current work in patterns uses a very low-level Gutenberg syntax. For example, a hero section becomes this combination of WordPress-specific commands and HTML styled with WordPress-provided classes:

```
<!-- wp:group {"backgroundColor":"very-light-gray","align":"full"} -->
<div class="wp-block-group alignfull has-very-light-gray-background-co<di
<!-- /wp:spacer -->

<!--</div><!-- /wp:spacer -->

<!-- wp:columns {"align":"wide"} -->lignwide"><!-- wp:column -->
<div class="<!-- wp:column --><!-- wp:column -->
<div class="wp-block-column"><!-- wp:heading {"textColor":"very-dark-gray
<!-- /wp:heading -->

<!-- wp:paragraph {"textCol</h2><!-- /wp:heading -->

<!-- wp:paragraph {"textColor":"very-dark-gray","fontSize":"medium"} -->y
<!-- /wp:paragraph -->

<!-- wp:buttons -->
<div class</p><!-- /wp:paragraph -->

<!-- wp:buttons -->
<div class="wp-block-buttons"><!-- wp:button {"backgroundColor":"strong-b
```

```
<!-- /wp:button --></div>
<!-- /wp:buttons --></div>
<!-- </a><!-- /wp:button --></div>
<!-- /wp:buttons --></div>
<!-- /wp:column -->

<!-- wp:column -->
<div class="wp-block-column"><!-- wp:image {"id":306,"width":512,"height"
<!-- /wp:image --></div>
<!-- /wp:column --></div>
<!-- /wp:co<img src="https://nrqsnchztest.blog/wp-content/uploads/2020/02
<!-- /wp:column --></div>
<!-- /wp:columns -->

<!-- wp:spacer -->
<div style="height:100px" aria-hidden="true" class="wp-block-spacer"></di
<!-- /wp:spacer --></div></div>
<!-- /wp:group -->
```

and looks like this:

Lorem ipsum
consectetur a

Ut enim ad minim ven
ullamco laboris nisi ut
consequat.

Call to action

Figure 3: Hero Pattern for use with Gutenberg

Right now, designing patterns is not as easy or intuitive as it can be. I expect it to change when the code for patterns lands in the plugin and then in WordPress core. I also expect to be able to author the content visually using the built-in editor.

Links and Resources

- Block patterns: create patterns to start populating the library #20345
- Variations (formerly patterns) API for blocks #16283
- Block Patterns: Add ability for predefined block layouts to be added to a document #17335
- Docs: Add docs for variations in the block registration section #20145

Gutenberg as design systems (part 3)

Blocks are awesome and provide a graphical way to create content but sharing them is not as intuitive as I would like to be, at least not yet.

This post will discuss how to use plugins to share both custom blocks, variations and combined blocks and variations.

Custom Blocks

In the plugin's `index.php` we create a block category where we can place all our blocks.

The comment at the top of the page contains the information that will appear in the plugins page.

The first functions `rivendellweb_blocks_block_category` defines a callback function that creates a category for the blocks we create. We then use the `block_categories` filter with the `rivendellweb_blocks_block_category` callback function.

Next we include the blocks that we have creaated using `require example-0x/index.php` where the x represents a different directory containing a custom block.

```
<?php

/**
 * Plugin Name: Rivendellweb Blocks
 * Plugin URI: https://github.com/WordPress/rivendellweb-blocks
 * Description: Rivendellweb blocks collection.
 * Version: 0.0.1
 * Author: Carlos Araya
 *
 * @package rivendellweb-blocks
 */

if ( ! defined( 'ABSPATH' ) ) {
  exit;
}

function rivendellweb_blocks_block_category( $categories, $post ) {
  if ( $post->post_type !== 'post' ) {
      return $categories;
  }
  return array_merge(
    $categories,
    array(
      array(
        'slu'ABSPATH'vendellweb-blocks',
        'title' => __( 'Rivendellweb Blocks', 'rivendellweb-blocks' ),
        'icon'  => 'wordpress',
      ),
```

```
    )
  );
}
add_filter( 'block_categories', 'rivendellweb_blocks_block_category', 10,


include 'example-01/index.php';
include 'example-02/index.php';
include 'example-03/index.php';
include 'example-04/index.php';
include 'example-05/index.php';
include 'example-06/index.php';
include 'example-07/index.php';
```

If the blocks are valid and the individual build processes succeeded, then the plugin will add seven blocks to Gutenberg running on the server.

The full example is on github at rivendellweb-blocks and have been tested with Gutenberg 7.9.1 and with Github Gutenberg.

Block Variations

As discussed earlier, variations allow you to change the look of a block without having to code a brand new version. The plugin that holds these variations has three files:

- A PHP file that makes the package into a plugin and links to scripts and styles
- A Javascript files with the variation definitions
- A CSS files containing the variations' styles

The plugin's `index.php` has the plugin boilerplate comment and two action + callback instances: one to enqueue the scrip and one to enqueue the stylesheet.

```
<?php
```

```
/**
 * Plugin Name: Rivendellweb variations
 * Plugin URI: https://github.com/WordPress/rivendellweb-variations
 * Description: Rivendellweb blocks variations.
 * Version: 0.0.1
 * Author: Carlos Araya
 *
 * @package rivendellweb-blocks
 */

if ( ! defined( 'ABSPATH' ) ) {
  exit;
}

function rivendellweb_enqueue_variations() {
    wp_enqueue_script(
        'rivendellweb-script',
        plugins_url( './src/block-variations.js', __FILE__ ),
        array( 'wp-blocks', 'wp-dom-ready', 'wp-edit-post' 'ABSPATH'  fi
    );
}
add_action( 'enqueue_block_editor_assets', 'rivendellweb_enqueue_variatio

function rivendellweb_variation_styles() {
    wp_enqueue_style(
        'rivendellweb_variations_css',
        plugins_url( './src/block-variations.css', __FILE__ ) );
}
add_action( 'enqueue_block_assets', 'rivendellweb_variation_styles' );
```

The full example is on github at rivendellweb-variations and have been tested with Gutenberg 7.9.1 and with Github Gutenberg.

Both Together

Whether to combine the two plugins discussed in this post into a single plugin is

the reader’s choice. I chose not to because I believe that each plugin should do one thing only and do it well.