Building Gutenberg Blocks (Part 1)

Whether we like it or use it, the Gutenberg editor is here to stay. While I don't use it for my own content and have developed themes that rely on the classic editor plugins, I realize that any WordPress shop moving forward needs to at least be aware of how to build blocks.

It's unlikely that the Classic Editor plugin or other means to disable Gutenberg will remain available past 2022 (when the Classic Editor plugins is supposed to stop being fully supported).

So this post will cover the creation of Gutenberg blocks and some basic setup in both PHP and Javascript

Creating the build system

The following block examples are meant to be built inside a plugin. They won't work if installed inside a theme, which is also not recommended and will result on your theme being rejected for inclusion in the plugin directory.

PHP setup

On the root of the plugin folder we'll add a main PHP file to be the driver for all the blocks that we will create.

The first part is the plugin metadata. The information here will allow WordPress to use the plugin.

The name is required, everything else is optional.

```
<?php
/*
Plugin Name: rivendellweb-blocks
Description: Collection of blocks based
  on work done on the Rivendellweb
  theme
```

```
Version: 0.0.1
Author: Carlos Araya
Author URI: https://example.org
*/
```

The second part block is the actual code.

Exit the page unless there is a defined ABSPATH constant.

If we haven't exited then require the `index.php` for each block that we want to work with. This allows us to keep the code for each block separate and the code inside the plugin

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

include 'rivendellweb-static/index.php';
```

Setting up the build environment

Because I've chosen to write my blocks in ESNext I have to setup a Node-based build system.

The first thing to do is to create a `package.json` file to store the plugins.

```
npm init --yes
```

This will create the package file and populate it automatically with all my configured pereferences.

Next, we need to create a `.gitignore` file so that we won't push unnecessay files and folders to git. To do this we run the following command

```
npx gitignore node
```

This will run npx, part of Node as of 5.2, and run the gitignore package for Node projects.

Instead of configuring WebPack to build and bundle the code directly, I will use `@wordpress/scripts` as the tooling for the project.

```
npm i -D @wordpress/scripts
```

Once we have the package installed, we need to replace the scripts block with one configured to use the scripts in the package. Modify the `scripts` portion of package.json to use the following scripts:

```
`"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"
}`
```

Building Blocks

With the basic building tools in place we'll concentrate on building the blocks themselves inside a plugin.

We'll build a base block and then enhance it with additional functionality.

Each block has two main files: `index.php` that manages the communication with WordPress core and `index.js` that runs the react code we need to make the block display in Gutenberg and render on the page.

All the block examples require you to build them using the process outlined in the previous section. They will not work otherwise.

You can check a finished version of the blocks in the Github repository at https://github.com/caraya/rivendellweb-blocks

Base

The first block will display static content like we might do for a call to action or other static content.

PHP

In the PHP side, the first part makes sure that we're not trying to access the script directly by checking if the constant ABSPATH is defined. If not, we exit and do nothing.

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

If we pass the check then we start working with internationalization.

We use the load_plugin_textdomain to load a plugin's translated `.mo` files.

We then add the function to the init hook.

```
function rivendellweb_blocks_example_01_load_textdomain() {
  load_plugin_textdomain( 'rivendellweb-blocks', false, basename( __DIR__
}
add_action( 'init', 'rivendellweb_blocks_example_01_load_textdomain' );
```

The second part is a little more complex.

We first test to see if Gutenberg is supported by checking if `register_block_type` is supported. If it's not then we return as there is nothing for us to do.

It uses `index.asset.php`, a file generated during the build process, to reference pre-requisites and other assets to set them to a variable.

`wp_register_script` will register a script and its dependencies to be enqueued at a later time. It is different than using `wp_enqueue_script`.

`register_block_type` does the bulk of the work, it registers the block type and makes it available to the block editor.

The last part of this function is to check if `wp_set_script_translations` is available, meaning that we have translation tools available. If it is then we use it to load all the translated scripts for the plugin.

We also hook this function to the init hook to make sure that it's available when Wordpress first start.

```
function rivendellweb_blocks_example_01_register_block() {

  if ( ! function_exists( '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']
  );

    register_b'build/index.asset.php'b-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' );
```

Javascript / React

The Javascript portion of the block uses the `i18n` and `blocks` packages from WordPress to build the block.

If you've done work with internationalization you may recognize the `__()` command as command you use to internationalize strings.

We first import the components that we want to use.

For this block we'll define the styles inside a constant, `blockStyle`, that will be used for both the editor and the rendered content.

The block title uses a localized string to indicate that translators can work with it.

The icon is one of those available in WordPress dashicon library or a custom SVG.

The part that surprised me the most is the need for two statements with the same content. The `edit()` creates the content specific to the editor while `save()` works with content specific to what will render on the client.

The block will use the same style in the editor and the front end.

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

const blockStyle = {
```

```
  backgroundColor: '#639',
  color: '#fff',
  padding: '20px',
};

registerBlockType( 'rivendellweb-blocks/example-01', {
  title: __( 'Example 01', 'rivendellweb-blocks' ),
  icon: 'universal-access-alt',
  category: 'layout',
  example: {},
  edit() {
    return (
      <div style={ blockStyle }>
        <h2>Notice</h2>
        <p>Hello World, step 1 (from the editor).</p>
      </div>
    );
  },
  save() {
    return (
      <div style={ blockStyle }>
        <h2>Notice</h2>
        <p>Hello World, step 1 (from the front end).</p>
      </div>
    );
  },
} );
```

We could create different constants holding different styles and then use them to produce different results for the editor block versus what appears in the front end.

External Styles

SO far all the styles have been hardcoded to the block. In this example we'll make the block use an external stylesheet for styling the block.

To use external style sheets we first have to remove the constant containing the styles for the element.

We then add one or more calls to `wp_register_style` to add the stylesheets.

We also have to modify the call to `register_block_type` to acommodate the styles

- **`style`** is used for the default styles
- **`editor_style`** is used for editor styles if they are different than the default. This loads after the default styles so they will override the basiic styles if they are different.

```
<?php
// The editor styles override the default styles set below
wp_register_style(
  'rivendellweb-blocks-example-02-editor-stylesheets',
  plugins_u'rivendellweb-blocks-example-02-editor-stylesheets'cks' ),
  filemtime( plugin_dir_path( __FILE__ ) . 'editor.css' )
);

' ),
  filemtime( plugin_dir_path( __FILE__ ) . '// Default styles
wp_register_style(
  'rivendellweb-blocks-example-02',
  plugins_url( 'style.css', __FILE__ ),
  array( ),
  filemtime( plugin_dir_path( __FILE__ ) . 'style.css' )
);

register_block_type( 'rivendellweb-blocks/example-02-stylesheets', array(
  'style' => 'rivendellweb-blocks-example-02',
  'editor_style' => 'rivendellweb-blocks-example-02-editor-stylesheets',
  'editor_script' => 'rivendellweb-blocks-example-02',
) );
```

Building Gutenberg Blocks (Part 2)

We've only worked with static content so far. But the idea behind Gutenberg is to let you work with your own content.

This section will also introduce the `RichText` block components as opposed to the plain text elements we've been working with.

Editable Blocks

The first part of this process is to create an editable block where we can enter arbitrary data that will be saved when we save or publish the post.

IN addition to i18n and blocks imports we import the RichText component from the block-editor package. This will do the heavy load in the code below.

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

The first difference is that we've now added an `attributes` attribute that will hold the structured data needs of a block. Wordpress will also use this to validate the content of the block.

In this example the `content` child of attribute is an array of children that paragraphs (the children match the p selector).

```
registerBlockType( 'rivendellweb-blocks/example-03', {
    title: __( 'Example 03', 'rivendellweb-blocks' ),
    icon: 'universal-access-alt',
    category: 'layout',
    attributes: {
        content: {
            type: 'array',
            source: 'children',
```

```
            selector: 'p',
        },
    },
```

The edit and save methods get a little moore complicated when we use custom data.

The `save()` method now takes props as an attribute.

We assign the content attribute, the setAttribute and className to props.

Next, we create an onChangeContent function that will trigger when we add or remove cotnent from the element.

Finally, we return a `RichText` element that returns p elements with the `className` class and the `content` as the value of the element.

The onChange event firest with the `onChangeContent` every time the element changes.

```
    example: {},
    edit: ( props ) => {
        const {
            attributes: { content },
            setAttributes,
            className,
        } = props;

        const onChangeContent = ( newContent ) => {
            setAttributes( { content: newContent } );
        };

    return (
            <RichText
                tagName="p"
                className={ className }
                onChange={ onChangeContent }
                value={ content }
```

```
            />
        );
    },
```

The `save()` method also takes `props` as a parameter and uses `props.attributes.content` as the value of the `value` parameter.

```
    save: ( props ) => {
        return (
            <RichText.Content
                tagName="p"
                value={
                  props.attributes.content
        } />
        );
    },
} );
```

Inner blocks

The next idea is to see if we have any way to automate the content of the block. Right now our block is limited to whatever tag we use as the value of `tagName`.

We can create a single block with a list of default blocks using inner blocks.

We import `InnerBlocks` from `@wordpress/block-editor` and then use it as the child of our content div.

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

'@wordpress/block-editor'// Register block
registerBlockType( 'rivendellweb-blocks/example-04', {
    title: 'Example 04',
    category: 'rivendellweb-blocks',
    icon: 'translation',
```

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

The idea is that using Inner Blocks we get a set of blocks available without any additional code.

Blocks with customized content content

Inner blocks give us a set of sensible default set of blocks to use but it may or may not be enough to suit our needs. Fortunately we don't have to stick with the defaults if we need specific blocks to accomplish our goals.

First we run all our imports as usual, __ for i18n, `registerBlockType` and `InnerBlocks`

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

THe first difference is the inclusion of an `ALLOWED_BLOCKS` constant that we'll use later to tell Gutenberg what blocks to allow in our inner blocks.

```
const ALLOWED_BLOCKS = [
  'core/image',
```

```
  'core/heading',
  'core/paragraph'
];
```

The final change is in the `edit` method of registerBlockType where we add an `allowedBlock` attribute and use the `ALLOWED_BLOCKS` constant as the value.

```
registerBlockType( 'rivendellweb-blocks/example-05', {
  title: __('Example 05', 'rivendellweb-blocks'),
    category: 'rivendellweb-blocks',
    icon: 'translation',

  edit: ( { className } ) => {
    return (
      <div className={ className }>
        <InnerBlocks
          allowedBlocks = { ALLOWED_BLOCKS }/>
      </div>
    );
  },

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

The idea is that we restrict the content of the InnerBlock element to whatever elements we want to use. This way we can be sure that users don't modify the content in ways we don't expect.

Block Templates

We can go even further and define a complete template for the block, not only limiting the blocks but providing a ready-to-complete template

The import elements don't change. We import __, `registerBlockType` and `InnerBlocks` as before.

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

Instead of an allowed blocks section we create a `BLOCK_TEMPLATE` variable that list the blocks we want to use, the order of the blocks and, where appropriate, placeholdeer text.

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

The template attribute uses our `BLOCK_TEMPLATE` constant as the layout of the blocks we're building

`templateLocks` allows locking the InnerBlocks area for the current template. It takes one of the following values:

- `all` — prevents all operations. It is not possible to insert new blocks. Move existing blocks or delete them.
- `insert` — prevents inserting or removing blocks, but allows moving existing ones.
- `false` — prevents locking from being applied to an InnerBlocks area even if a parent block contains locking.

For example, our book block may want to do more than one paragraph summary or we may want to add more images. Or maybe not, the choice is up to the designer.

```
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="false" />
      </div>
    );
  },

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

We can build multicolumn blocks using core blocks and place them in a template. The following example creates a two column layout with an image in the first column and one or more paragraphs in the second.

```
const TEMPLATE = [
  [ 'core/columns', {}, [
    [ 'core/column', {}, [
        [ 'core/image' ],
    ] ],
    [ 'core/column', {}, [
        [ 'core/paragraph', { placeholder: 'Enter side content...' } ],
    ] ],
  ] ]
];
```

So now we have the flexibility of building composite blocks with pre-defined structure and content flow.

Building Gutenberg Blocks (Part 3)

Internationalization

We'll take a break from building actual blocks and talk about internationalization or i18n. Taking good care of i18n when you build your plugin will allow you or someone else to translate the text of your plugin so it's easier to use in languages other than the one it was written on

It works very close to how the PHP i18n functions work (and I believe this is deliberate), just differently enough to take the Javascript needs into account.

First we import the necessarty functions. Most of the time it'll just be __ but there are others.

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

Now, whenever we have a string that we want to translate, we put inside the __ command and indicate the text domain it's associated with.

This example tells translators that it's the string to translate is `Example 07` and the text domain is `rivendellweb-blocks`.

```
title: __( 'Example  07', 'rivendellweb-blocks' )
```

Once we have all the strings ready for translation we need to generate a POT file to translate from and a PO file with the translated strings.

Translating plugins is similar to how you translate themes. I've documented the process in Translating WordPress Themes so I won't go into too much detail about it here.

Using Poedit the process is simple when using the Pro or Pro+ version.

1. Open Poedit and select Translate WordPress theme or plugin

2. Locate the plugin folder
3. Generate a POT file
4. Save the POT file to a `languages` folder inside your plugin. Create it if it doesn't exist
5. Use the POT file to generate your translation
6. Save the .po translation file to a `languages` folder inside your plugin

The last step is to load all available translations for our plugin from the languages directory by using load_plugin_textdomain() like the example below.

```php
<?php
function rivendellweb_blocks_example_07_load_textdomain() {
    load_plugin_textdomain(
        'rivendellweb-blocks',
    false,
    basename( __DIR__ ) . '/languages' );
}

add_action( 'init', 'rivendellweb_blocks_example_07_load_textdomain' );
```

So now we've made our plugins fully translatable and if we add new functionality or update the wording of the plugin text shown to the user we just need to update the template and the translations.

Building Gutenberg Blocks (Part 4)

There are many other things we can do with blocks and things to explore. But for right now we'll move into enhacing the blocks we've built and add some flare to them.

Making the editor match the front end

Before we look at some enhancements for the blocks, let's make sure that the editor matches the front end content.

The match will not be 100%; The way I've set up my theme is different than the way blocks are styled in Gutenberg, but it's close enough to make it not look very disimilar.

The following commands are included in an init function and then attached to the `after_setup_theme` action.

The first block adds editor-specific styles and the default styles for the built-in blocks.

```
<?php
add_theme_support('editor-styles');
add_editor_style( '/editor-styles.css' );
add_theme_support( 'wp-block-styles' );
```

The next block sets custom font sizes for the editor dialogues. Whenever Gutenberg presents you with a font-size selection dialogue, these are the values it will use.

I've trumcated the list to make the post easier to read. The full list has three additional values.

```
<?php
add_theme_support(
'edit'editor-font-sizes'ay(
  array(
      'name' => __( 'Small', 'rivendellweb-blocks' ),
 'name'size' => 10,
      'slug' => 'small'
  ),
  array(
      'name' => __( 'Regular', 'rivendellweb-blocks' ),
      'size' => 16,
' => 10,
      ''regular'
  )
  'regular'// truncated to save space
);
```

Like we did with the custom fonts sizes we do the same thing with a custom color palette.

```
<?php
add_theme_support(
'edit'editor-color-palette'ay(
  array(
      'name' => __( 'Magenta', 'rivendellweb-blocks' ),
  'name'lug' => 'magenta',
      'color' => '' => '#a156b4',  array(
      'name' => __( 'Light Magenta', 'rivendellweb-blocks' ),
      'slug' => 'light-magenta',
      'color' => '#d0a5d'name'#d0a5db',
  ),
  // Truncated to save space
)
);
```

The idea is that the theme defines the colors we use and then Gutenberg picks them up automatically and uses them as if they were part of the default.

Adding Custom Block Categories

In addition to preset categories for blocks, we can also create our own categories to hold blocks.

We had the block class in PHP using the `block-categories` filter.

The blocks are meant for posts only so we first make sure that we are working with posts before we proceed, if we're not then we return the existing categories

```
<?php
function rivendellweb_blocks_block_category( $categories, $post ) {
  if ( $post->post_type !== 'post' ) {
    return $categories;
  }

  return array_merge(
    $categories,
    array(
      array(
        'slug' => 'rivendellweb-blocks',
        'title' => __( 'Rivendellweb Blocks', 'rivendellweb-blocks' ),
        'icon'  => 'wordpress',
      ),
    )
  );
}
add_filter( 'block_categories', 'rivendellweb_blocks_block_category', 10,
```

See Gutenberg: Creating Custom Block Categories for more information

Building Gutenberg Blocks (Part 5)

While we can do some basic formatting with the blocks, we can also add settings to the block and give users a more direct means of customization

Block Toolbars and Inspectors

Gutenberg blocks have 2 different configurations: toolbars and inspectors.

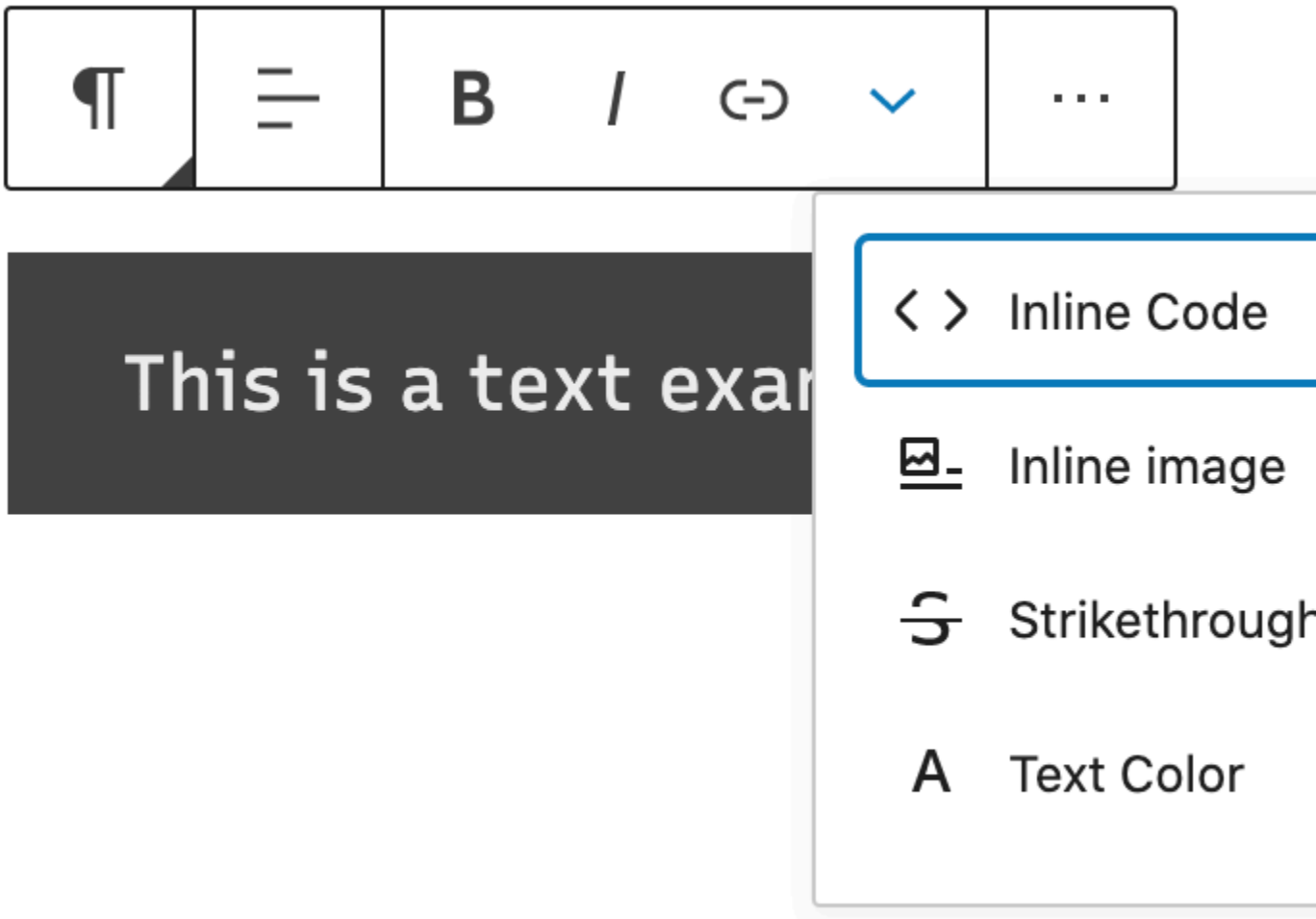

Figure 1: Guttenberg paragraph toolbar showing top level and pull down options

Toolbars appear above the block we're working on and provide additional functionality.

Note that BlockControls is only visible when the block is currently selected and

in visual editing mode. It will not work if you’re editing the block in HTML mode.

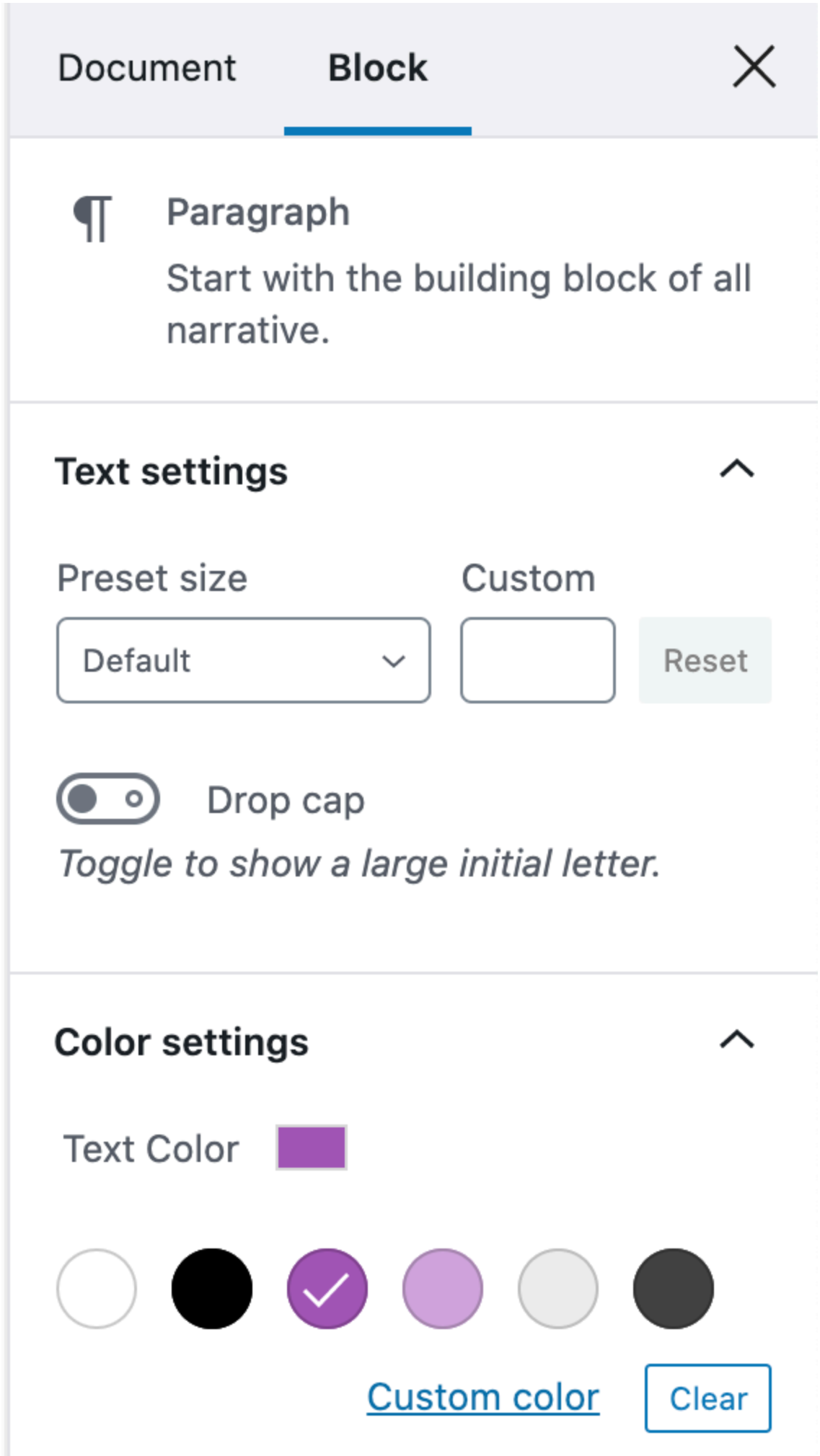
Document
Block
Paragraph
Start with the building block of all narrative.
Text settings
Preset size
Default
Custom
Reset
Drop cap
Toggle to show a large initial letter.
Color settings
Text Color
Custom color
Clear

Figure 2: Guttenberg inspector sidebar showing multiple option panels

The inspector provides additional functionallity to the block and functionality that is not essential for the block to function.

The Settings Sidebar is used to display less-often-used settings or settings that require more screen space. The Settings Sidebar should be used for block-level settings only.

If you have settings that affects only selected content inside a block (example: the **bold** setting for selected text inside a paragraph): do not place it inside the Settings Sidebar.

The Settings Sidebar is displayed even when editing a block in HTML mode, so it should only contain block-level settings.

The Block Tab is shown in place of the Document Tab when a block is selected.

Similar to rendering a toolbar, if you include an InspectorControls element in the return value of your block type's edit function, those controls will be shown in the Settings Sidebar region.

THis works awesome for core components, but how do we make it work for custom blocks or those with special requirements?

I've created a blank component to test the toolbar and sidebar items and see how they work. The toolbar works flawlessly but the Inspector doesn't seem to work at all.

The PHP file doesn't change much.

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

```
function rivendellweb_blocks_load_textdomain() {
    load_plugin_textdomain( 'rivendellweb-blocks', false, basena'rivendel
}
add_action( 'init', 'rivendellweb_blocks_load_textdomain' );

function rivendellweb_blocks_register_block() {
    ' );
}
add_action( '// automatically load dependencies and version
    $asset_file = include( plugin_dir_path( __FILE__ ) . 'build/index.asse

    wp_register_script(
        'rivendellweb-blocks-example-07',
        plugins_url( 'build/index.js', __FILE__ ),
        $asset_file['dependencies'],
        $asset_file['version']
    );

    register_block_type( 'rivendellweb-blocks/example-07', array(
        'editor_script' => 'rivendellweb-blocks-example-07',
    ) );
  add_action( 'init', 'rivendellweb_blocks_register_block' );


  if ( function_exists( 'wp_set_script_translations' ) ) {
    wp_set_script_translations( 'rivendellweb-blocks-example-07', 'gutenbe
  }

}
```

Links and Resources

- Get Started With Gutenberg Development
- Build for Gutenberg: How Plugin and Theme Authors Are Addressing the Transition to Gutenberg

- Generate Blocks with WP-CLI
- Gutenberg at VIP
- A Gutenberg Tutorial for Beginner Developers: Create Your First Block Plugin
- create-guten-block
- Blocks Tutorial
- Internationalization in WordPress 5.0
- gutenberg-examples
- how-to-gutenberg-plugin
- classnames React packages
- How to add WordPress Theme Styles to Gutenberg
- Block Style Variations