Revisiting Gutenberg full site editing

Now that WordPress 5.9 is close to release, we can revisit the Gutenberg full site editing experience since more of it will be baked into core rather than the Gutenberg plugin.

I'm not a fan of full site editing just like I'm not completely sold on Gutenberg as an editor even after a few years. But, since the market is moving in this direction, I believe that developers should be aware of Gutenberg and the FSE to provide guidance on the optimal approach to the client's project.

I've been reading and researching the latest improvements to the FSE experience and I'm not certain if the current use cases have been thought through in Gutenberg.

Understanding the Full Site Editing experience

I've set up a brand new WordPress site to work with Gutenberg. I've documented previous work with Gutenberg in the blog posts below:

- Gutenberg full-site editing and Block-Based Themes
- A New Way to Create Block Plugins
- Gutenberg: A step forward or two steps back?
- Gutenberg: How do we work with older content?
- Gutenberg: Additional Thoughts and Conclusions
- Building Gutenberg blocks
    - Part 1
    - Part 2
    - Part 3
    - Part 4

Linking to the above gives me a baseline for the content in this post.

If you're reading this, I'm assuming can:

- Build React-based Gutenberg blocks

- Pack Gutenberg blocks into a plugin
- Create style variations for a block
- Style blocks with CSS
- Create a theme.json global configuration file

There are still some questions to ask. Having the answers would make moving to a Gutenberg-based theme easier:

- How can you load prism scripts and styles into a theme.json file?
- How do you include third-party fonts and scripts into a Gutenberg theme? is enqueueing the font enough?

Creating blocks versus the current system: React versus PHP

React blocks are the core of Gutenberg, both the ones that are bundled with WordPress Core, those that are built by third-party groups like StudioPress, and those that you build yourself as bespoke blocks that address your specific needs.

That said, Gutenberg doesn't fully eliminate the need for PHP. There are still places where PHP is necessary.

Block Filters : WordPress provides a series of filters that can be used to modify the block before it is rendered. : Some of the filters are written in PHP and the ones that seem to be written in Javascript, it is hard to tell from the examples and the prose surrounding them.

Plugins : plugins are still the preferred way to package content for use in WordPress, whether it's in Gutenberg or outside. : Writing plugins is not complicated but it's not trivial either. It gets more complicated if you plan to share the plugin in the WordPress.org repository as there are more rules to follow.

That brings up what, to me, is the biggest problem with Gutenberg:

> **Not all WordPress developers are React developers, and they shouldn't be**.

I understand that there are tons of blocks available by default and that several theme framework makers like Elementor and Genesis have released block plugins.

But just like with jQuery and any other framework, there is a time when your needs may grow beyond what's available, or beyond the price you're willing to pay, so you will have to build your own so yes, you will become a React developer.

A good starting point is the Node-based create-block package. This tool will automate the creation of a basic block for you but that's all it does, it's still on you as the developer to create the block itself. It will also start you on what the WordPress team considers best practices for creating blocks.

There are some new things that replace techniques that we've taken for granted as WordPress developers but they mean that we have to move away from the PHP that we've used since the beginning and move to React.

We'll discuss some of these features (as I understand them) in future posts.

New Gutenberg Features: Query blocks

Gutenberg offers the Query Block as an alternative to the standard loop written in PHP to populate and customize the site's content.

This is how the loop looks on PHP

```
<?php
$args = array(
  'post'posts_per_page',
  'post_type' => 'post',
  'post_status' => 'p'post_type'

$my_custom_query = new WP_Query( $args );

// Custom Loop
if( $my_custom_query->have_posts() ){
  while( $my_custom_query->have_posts() ){
    $my_custom_query->the_post();
    // HTML structure of our post
  }
} else {
    // Nothing matched the query
}

// Return to regular query loops
wp_reset_postdata();
```

The query and query loop blocks provide equivalent functionality when working with Gutenberg both to populate and to modify the results of the query.

To add a block click on the add block button and select the `Query Loop block`. You can also type "`/query loop`" and hit enter in a new paragraph block to add one quickly.

query

Query Loop

Posts List

Archive Title

Comments Query Loop

Standard

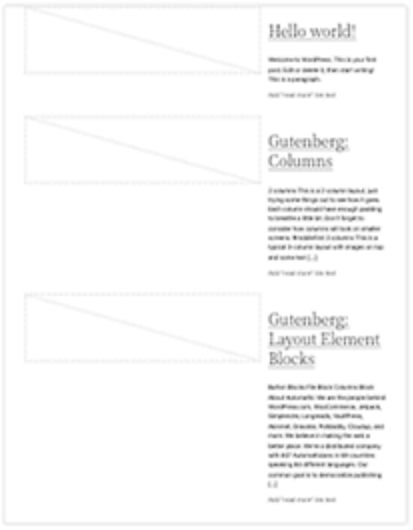

Image at left

fse

Hello w

Welcome to WordP

This is a paragraph

Add "read more" li

Figure 1: Gutenberg add query loop dialogue with templates on the left and the option to start from scratch

Once you select the `query loo` from the block menu you're presented with a series of options below the block choices on the left side of the screen and the option to create a new post and start from scratch under the post settings on the right hand side.

If you choose to use one of the predefined templates then you can save and use the template.

If you choose to create your own then you will have to build the page from available or customized blocks and then publish it.

Think of the query loop block as providing the default structure for the posts. We can run additional query blocks to get comments, get posts from a specific category, tag, or author or geet custom post types.

Understanding the Query Block and Its Importance in Site Editing explains how to use the query block when desveloping a new theme.

We will discuss custom post types, how to create them in PHP and how to use them in Gutenberg in future posts.

New Gutenberg Features: Creating block and page templates

As part of the Full Site Editing (FSE) experience, templates and page templates give you a way to prepackage blocks (similar to what variations do) and full-page templates for the theme.

Templates and page templates currently require the Gutenberg plugin and will not run in WordPress Core right now, that will change sometime during the 5.9 cyclle.

To understand the block template and block template parts work let's look at the structure of a block-based theme.

```
.
├── LICENSE
├── README.md
├── assets
|   ├── css
|   ├── images
|   └── js
├── block-template-parts
|   ├── footer.html
|   └── header.html
├── block-templates
|   ├── 404.html
|   ├── index.html
|   ├── page-home.html
|   ├── page.html
|   └── single.html
├── functions.php
├── inc
|   ├── block-patterns.php
|   └── block-styles.php
├── index.php
```

```
├── readme.txt
├── screenshot.png
├── style.css
└── theme.json
```

The themes reorganize the structure as well as the way we author the content.

For contrast we'll look at the structure of Twenty Twentyone, an old default theme in WordPress.

The equivalent to the content in `template-parts` and its child directories in older themes like Twenty Twentyone is stored in a single `block-templates` directory.

The `block-template-parts` directory is a flattened version of the `template-parts` directory in Twenty Twentyone.

```
.
├── 404.php
├── archive.php
├── assets
|   ├── css
|   ├── images
|   ├── js
|   └── sass
├── classes
|   ├── class-twenty-twenty-one-custom-colors.php
|   ├── class-twenty-twenty-one-customize-color-control.php
|   ├── class-twenty-twenty-one-customize-notice-control.php
|   ├── class-twenty-twenty-one-customize.php
|   ├── class-twenty-twenty-one-dark-mode.php
|   └── class-twenty-twenty-one-svg-icons.php
├── comments.php
├── footer.php
├── functions.php
├── header.php
├── image.php
├── inc
```

```
|   ├── back-compat.php
|   ├── block-patterns.php
|   ├── block-styles.php
|   ├── custom-css.php
|   ├── menu-functions.php
|   ├── starter-content.php
|   ├── template-functions.php
|   └── template-tags.php
├── index.php
├── package-lock.json
├── package.json
├── page.php
├── postcss.config.js
├── readme.txt
├── screenshot.png
├── search.php
├── searchform.php
├── single.php
├── style-rtl.css
├── style.css
└── template-parts
    ├── content
    ├── excerpt
    ├── footer
    ├── header
    └── post
```

The other major difference is how the templates are written. Rather than using PHP and HTML, block templates use special HTML comments.

You can build templates directly in Gutenberg using the block editor. To edit with the built-in editor, follow these steps:

Select the editor under appearance menu. If you don’t see the editor link then you don’t have the latest version of the Gutenberg plugin as this is not part of WordPress Core.

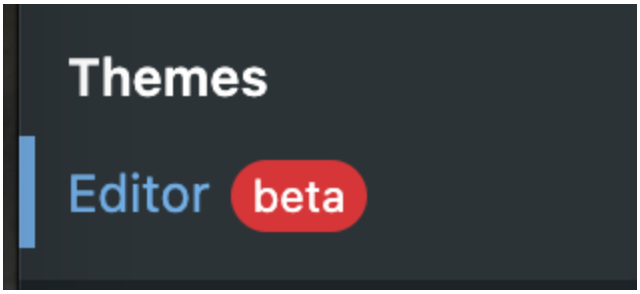

Figure 2: Gutenberg Editor Link Location

Once you are in the editor, click on the WordPress logo on the the top left of the screen and you will be shown the options to edit templates or template parts.

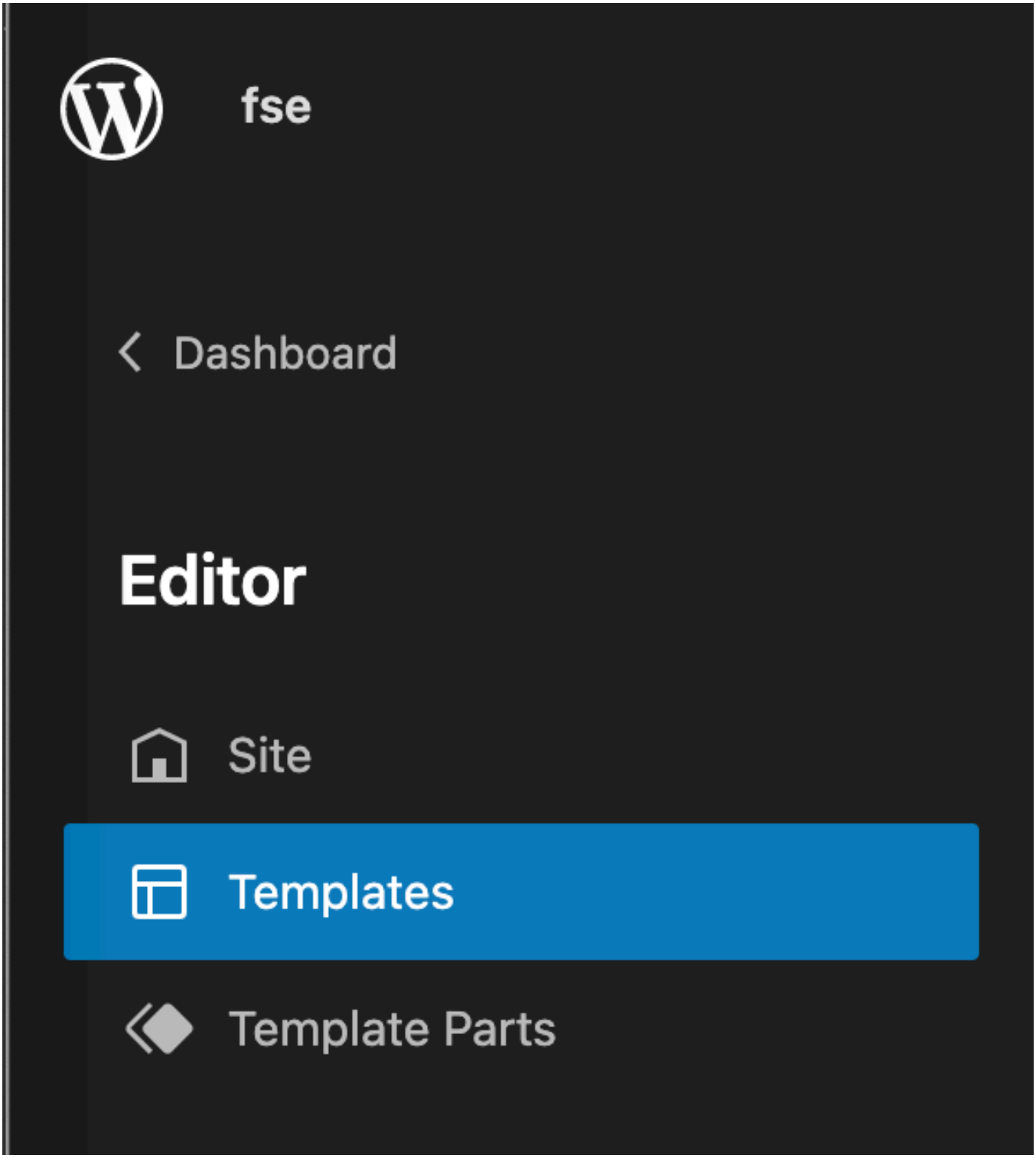

Figure 3: Gutenberg Editor Link Location

If you select the templates option you will see a list of available templates that you can already use on your page.

To create a new template click on Add New and give it a title. You can then edit the template as any other item in Gutenberg.

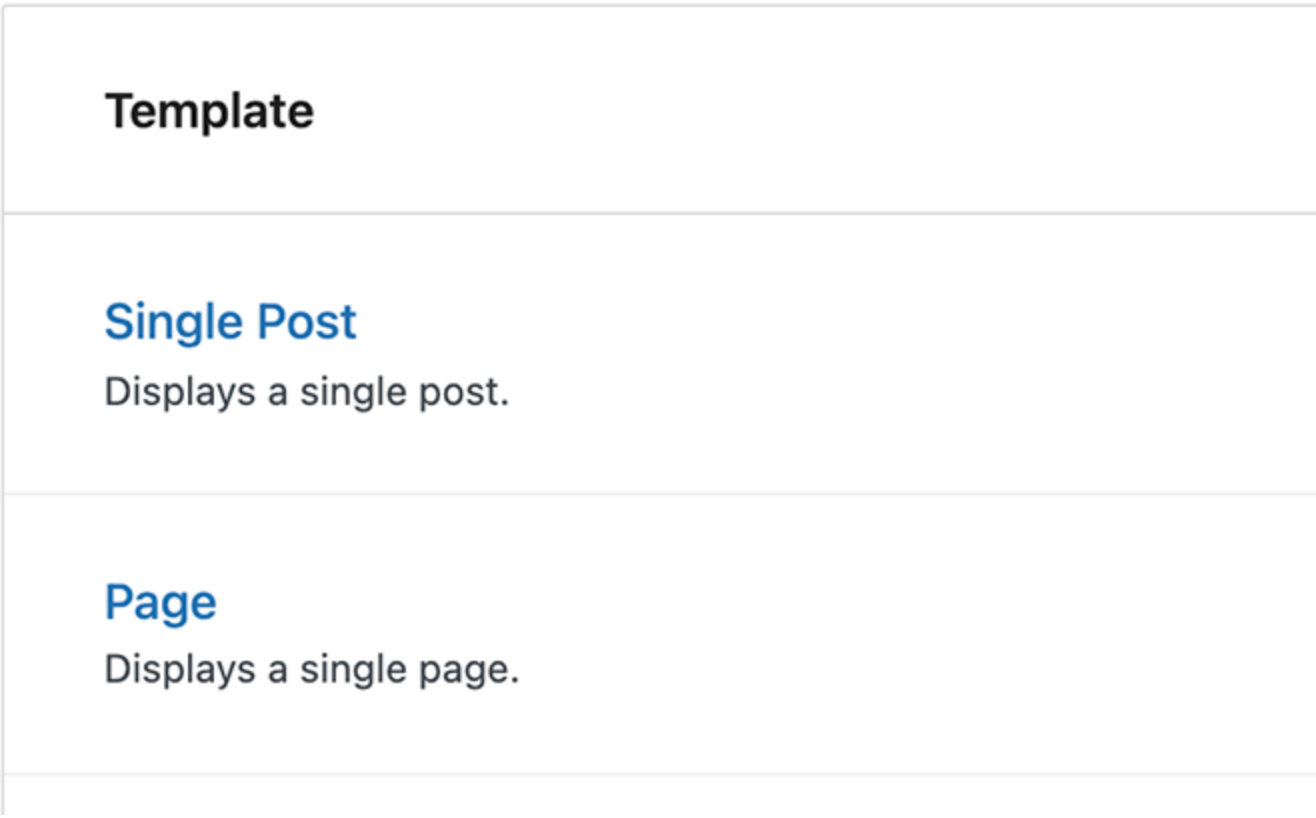

Figure 4: Add new page templates

Another way to edit templates is to manually write the code that Gutenberg uses to generate the code.

To do this first create a template that has all the elements and placeholders you want to use, then export the code by clicking the more options button on the top right of the screen and selecting 'export`. This will save a zip file with all the code on the templates to your local file system.

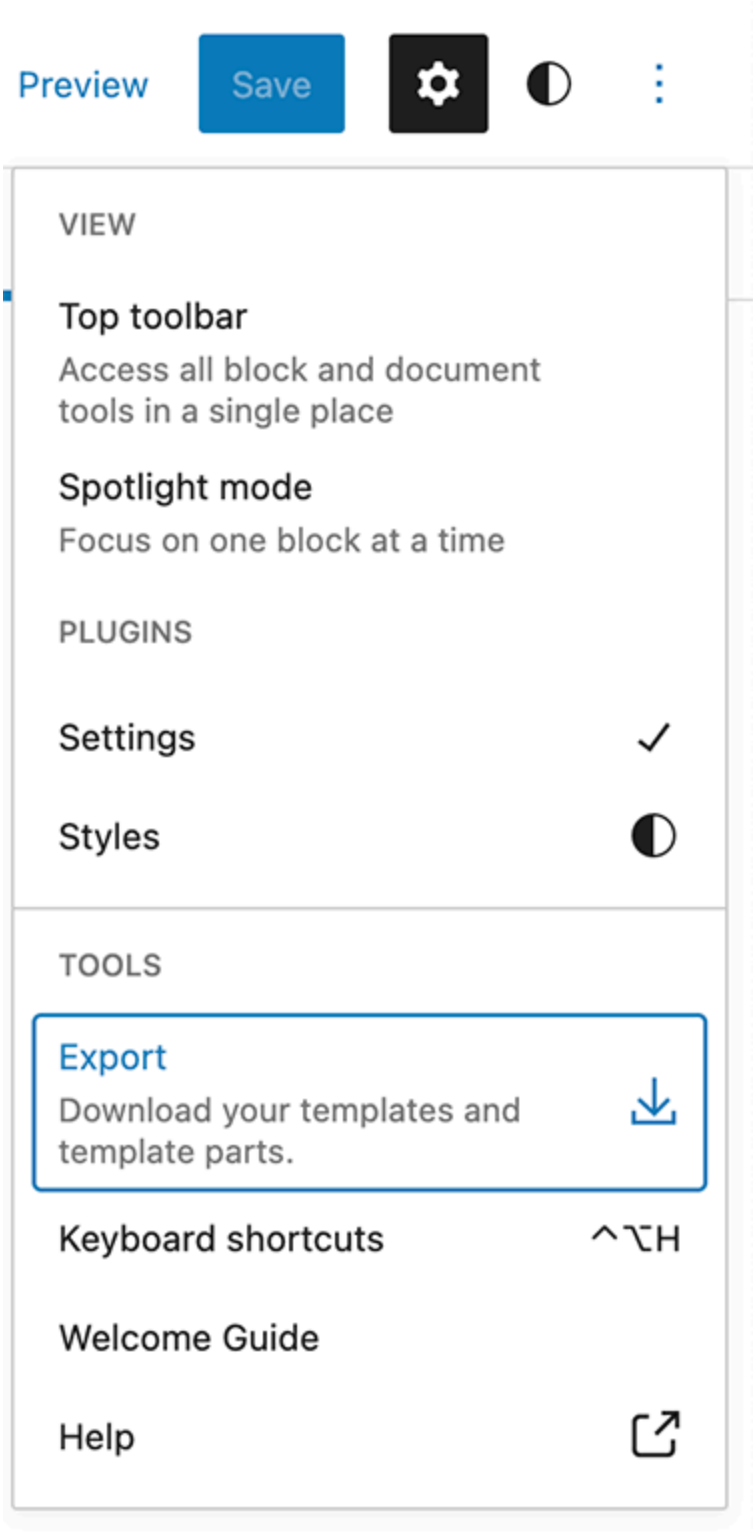

Figure 5: Export and download template and template parts

Warning

Some older material spoke about an edit option that would allow you to edit the template in place. I haven't been able to find it so I recommend the export method instead.

You can then copy the template files into your own templates or edit them to customize them. If you're customizing them it's important that you understand what the different parts of the templates do.

The templates use HTML comments and data with the wp: rivendellweb to indicate Gutenberg specific elements/blocks and curly brackets to indicate attributes.

The first example shows what a paragraph with attributes looks like. In this casse, the the attribute tells WordPress that the block is a paragraph and that it is aligned to the right. The content of the paragraph is the HTML and CSS necessary to render the right-aligned paragraph.

Note that for content that uses CSS, Gutenberg uses the has-{attribute name and value} convention for selector naming

```
<!-- wp:paragraph {"align":"right"} -->
<p class="has-text-align-right">... lik<p class="has-text-align-right">ed
</p><!-- /wp:paragraph -->
```

You can also nest blocks inside other blocks.

The next example shows the tags used for a query loop block along with pagination inside the individual posts.

```
<!-- wp:query -->
<div class="wp-bl<div class="wp-block-query"><!-- wp:post-template -->
  <!-- wp:post-title /-->
  <!-- wp:post-date /-->
  <!-- wp:post-excerpt /-->
```

```
  <!-- /wp:post-template -->

  <!-- wp:query-pagination -->
  <div class="wp-block-query-pagination">
    <!-- wp:query-pagination-previous /-->
    <!-- wp:query-pagination-numbers /-->
    <!-- wp:query-pagination-next /-->
  </div>
  <!-- /wp:query-pagination -->
</div>
<!-- /wp:query -->
```

The learning curve may seem daunting at first but, starting from existing templates, it’s not so daunting after all.

New Gutenberg Features: Conditionally loading block assets

One issue that I always found problematic with Gutenberg blocks is that it loads all assets all the time regardless of whether we use in our projects or not.

As of WordPress 5.8 we can work around this to only load the blocks that are used.

We define our block as normal. When creating a block we can define separate scripts and stiles for the front and back end using code like the one below:

```
<?php
register_block_type( 'rivendellweb/test-block',
  array(
    'editor_script' => 'rivendellweb-test-block-script',
    'editor_style'  => 'rivendellweb-test-block-editor-style',
    'style'         => 'rivendellweb-test-block-style',
    'script'        => 'rivendellweb-test-block-frontend'
  )
);
```

If you then also specify a filter that will only load the blocks assets when the block is actually used:

```
<?php
add_filter( 'should_load_separate_core_block_assets', '__return_true' );
```

For more reference, see: Conditionally Load Block Assets, Part 3 and Block-styles loading enhancements in WordPress 5.8

New Gutenberg Features: block and global configuration

Gutenberg provides additional ways to configure blocks and themes using JSON files one for block levels and one for global theme settings.

We'll look at each of these in more detail

block.json as a block configuration point

The `block.json` file is both a configuration point (what it is an how it works) and a way to make the metadata in the JSON file available to both PHP and Javascript.

The code below provides metadata for a block that creates message box for info, warning and danger notices.

```
{
  "$schema": "https://sche"https://schemas.wp.org/trunk/block.json""apiVe
  "title": "Not": ""title"tegory": "text": ""category" ["core/group"],
  ""core/group"",
  "description": "Shows info, ": ""icon": "star",
  "description"ords": ["alert", "message", ],
  "version": "": [""version""keywords""rivendellweb",
  "attribute"rivendellweb""version"      "type": "string",
 ": " "source": "html",
 ": ""source""attributes": {
    "message": {
      "type": "string",
      "source"sage": "message"": ""selector"ontext": ["groupId"],
  "s": [""providesContext": {
    "rivendellweb/message" { "name": "info", ": ""usesContext": ["groupId
  "supports""name": "warnin": ""label":  "War":  ""label""styles"": "dang
```

```
    { "name": "warning", "label": {
      "message": "": ""name": "danger", "label"
  "editorScript": "": ""name": "other", "label"cript": "file:./bui": ""exc
    "attributes": {
      "message"",
  "editorStyle": "file:./build/i": ""editorScript"e": "file:./build/style
": ""script": "file:./build/script.js",
  "viewScript": "file:./build/view.js",
  "editorStyle": "file:./build/index.css",
  "style": "file:./build/style.css"
}
```

If your theme supports lazy loading assets, then the enqueuing of scripts and styles for your block will be optimized and will only be loaded when the block is actively used. We discussed this technique in a previous post.

The Block Type REST API Endpoint can only list blocks registered on the server, so *the WordPress team recommends registering blocks server-side*. Using the `block.json` configuration file simplifies this registration.

If you wish to submit your block(s) to the Block Directory for inclusion, all blocks contained in your plugin must have a `block.json` file for the Block Directory to recognize them.

More information about the block directory can be found in this support article about the block directory.

If you choose not to submit your blocks to the block directory, the WordPress Plugins Directory can detect `block.json` files, highlight blocks included in plugins, and extract their metadata to show the users a list of the blocks bundled with your theme.

The `blocks.json` file follows a schema definition which makes development easier and allows other tools like text and code editors to provide validation, autocomplete and other support tools.

To use the schema, add the following to the top of the block.json.

```
"$schema": "https://schemas.wp.org/trunk/block.json"
```

For more information, check Block Metadata page in the Block Editor Handbook.

Block registration

One of the ways in which we can take advantage

PHP (server-side)

The register_block_type function that aims to simplify the block type registration on the server, can read metadata stored in the block.json file.

This function takes two params relevant in this context ($block_type accepts more types and variants):

- `$block_type (string)` – path to the folder where the block.json file is located or full path to the metadata file if named differently.
- `$args (array)` – an optional array of block type arguments. Default value: []. Any arguments may be defined. However, the one described below is supported by default:
    - **$render_callback (callable)**: callback used to render blocks of this block type.

It returns the registered block type (WP_Block_Type) on success or false on failure.

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

JavaScript (client-side)

Because the block is registered on the server, you only need to register the client-

side settings on the client using the same block's name.

```
registerBlockType( 'my-plugin/notice', {
    edit: Edit,
    // ...other client-side settings
} );
```

Registering the block on the server with PHP is still the recommended way to register the block, however, you can also register the block on the client using the `registerBlockType` method from the `@wordpress/blocks` package using the metadata loaded from `block.json`.

The function takes two parameters:

- **$blockNameOrMetadata (string|Object)** – block type name (supported previously) or the metadata object loaded from the block.json file with a bundler (e.g., webpack) or a custom Babel plugin.
- **$settings (Object)** – client-side block settings.

It returns the registered block type (WPBlock) on success or undefined on failure.

theme.json as a central configuration point

theme.json provides a means to configure a theme for use with Gutenberg that doesn't require editing `functions.php` or `style.css`.

The code below is a working theme.json and combines elements of the Armando theme by Carolina Nymark and a theme I'm working on to replace my existing Rivendellweb theme that runs my Publishing Project blog.

The first part of the block defines the schema location and version. I choose to use the `$schema` property to define the location of the schema so it's easier to work with the schema, having the URL present adds validation and syntax checking to the workflow in most editors.

Version 2 is the neweest version of the schema. Version 1 is still available but won't be receiving any further changes as far as I'm aware.

```
{
  "$schema": "https://sche"https://schemas.wp.org/trunk/theme.json""versi
```

The next section is the largest one and it configures all the settings for the theme. We will be able to override these default values in later definitions, otherwise, these are the value that the theme will use throughout.

If a property has a boolean (true/false) value it indicates if the theme will support the property, otherwise, the property will have one or more values attached to it.

Most of the properties will use three basci parameters:

- **slug** — the slug (computer-readable name) of the property
- **value / gradient** — the value of the property (it may also be an array of values)
- **name** — the human-readable name of the property

```
"settings": {
  "appearanceTools": true,
  "border": {
    "color": true,
    "radius": true,
    "style": true,
    "width": true
  },
  "color": {
    "background": true,
    "custom": true,
    "customDuotone": true,
    "customGradient": true,
    "defaultGradients": true,
    "defaultPalette": true,
    "link": false,
    "text": true,
    "duotone": [
      {
        "colors": [
```

```
            "#000",
            "#FFF"
          ],
          "#000""slug"ck-and-white",
          "name": "Bla": "nd White"
        }
      ],
      "gradients": [
      "gradients"   "slug": "blush-bordea": ""slug"     "gradient": "linea
          "name": "Blush bordeaux"
    ": ""name"     {
          "slug": "blush-light-purple",
   ": ""slug"adient": "linear-gradient(135deg,r": ""gradient") 0%,rgb(152
          "name": "Blush light purple"
        }
  ": ""name"   "palette": [
        {
          "slug": "str": ""palette"
          "color": "#a1": ""slug"       "name": "Strong magenta": ""color"
          "name" "very-dark-grey",
          "color": "rgb(131, 12"very-dark-grey""slug"": "Very dark grey"
        }
": ""color": "rgb(131, 12, 8)",
          "name": "Very dark grey"
        }
      ]
    },
```

Properties under “custom” create new CSS Custom Properties that we can use in other parts of the `theme.json` file and elsewhere in our CSS

The algorithm to create CSS Variables out of the settings under the “custom” key works this way:

We want a mechanism to parse back a variable name such `--wp--custom--line-height--body` to its object form in theme.json. We use the same separation for presets.

A few notes about this process:

- camelCased keys are transformed into its kebab-case form, as to follow the CSS property naming schema. lineHeight is transformed into line-height
- Keys at different depth levels are separated by –
- Don’t use – in the names of the keys within the custom object

```
    "layout": {
      "contentSize": "800px",
      "800px""wideSize": "1000px"
    },
    "spacing": {
      "blockGap": null,
      "padding": true,
      "margin": true,
      "units"     "px",
        "em",
        "rem",
        "vh",
        "vw"
      ]
    },
"px"
```

Typography controls typographical and font-related settings. This is where you define the font stacks for your theme. One outstanding item that I’m not sure how to handle is using web fonts you own rather than fonts from Google Fonts or other providers.

```
    "typography": {
      "customFontSize": true,
      "dropCap": true,
      "fontStyle": true,
      "fontWeight": true,
      "letterSpacing": true,
      "lineHeight": false,
      "textDecoration": true,
      "textTransform": true,
      "fontFamilies": [
        {
```

```
          "fontFamily": "-apple-system,BlinkMacSystemFont,\"Segoe UI\",Rob
          "-apple-system,BlinkMacSystemFont,\"Segoe UI\",Roboto,Oxygen-San
          "name"lug": "geneva-verdana"
        },
        {
    ": ""fontFamily"y": "Cambria, Georgia, serif",
          "slug": "c": ""slug": "geneva-verdana"
        },
        {
          "fontFamily"    "slug": "extra-small",
          "s": ""slug"x",
          "name": "Extra small"
        }": ""fontSizes"       "slug": "small",
": ""slug""size": "18px",
          "": ""size": "16px",
          "name"          "slug": "normal",
          "size": "": ""slug"       "name": "Norma": ""size": "18px",
          "name"lug": "large",
          "size": "24px",
": ""slug""name": "Large"
      ": ""size": "20px",
          "name"tra-large",
          "size": "40px",
    ": ""slug"e": "Extra large"
   ": ""size": "24px",
          "name""huge",
          "size": "96px",
       "huge""slug": "extra-large",
          "size": "40px",
          "name": "Extra large"
        },
        {
          "slug": "huge",
          "size": "96px",
          "name": "Huge"
        }
      ]
    }
```

```
  },
```

The declarations inside the `styles` object are applied to the body of the pages on the theme. Note that is uses CSS variables with the `--` separator as required for WordPress created CSS variables.

```
  "styles": {
    "color": {
      "background": "var(--wp--preset--color--white)",
      "var(--wp--preset--color--white)""text")"
    },
    "typography": {
      "fontSize": "20px",
   ": ": {
      "fontSize": "20px",
      "fontFamily"tem-fonts)",
      "lineHeight": "1.7"
    },
    "spaci": ""lineHeight""lineHeight"     "top": "0px",
": ""spacing"t": "0px",": ""margin"ttom": "0px"": ""top" "left": "0px"
  ": ""right": "0px",
        "bottom": "0px",
        "left": "0px"
      }
    },
```

Styles specified in the `elements` sectioon apply to HTML elements.

```
    "elements": {
      "link": {
        "color": {
          "text": "var(--wp--preset--color--dark-blue)"
        }
      },
      "var(--wp--preset--color--dark-blue)""h1"t": "var(--w": ""color": {
          "text"       },
```

```
      "typography": {
        "fontSize": "va": ""typography": {
        "fontSize"     }
    },
    "h2": {
      "color": {
        "text": "var": ": {
      "color": {
        "text",
      "typography": {
        "fontSize": "var(--wp--": ""typography""typography": {
        "fontSize""h3": {
      "color": {
        "text": "var(--wp--preset--c": ""h3": {
      "color": {
        "text": {
      "color": {
        "text": "var(--wp--preset--color": ""h4": {
      "color": {
        "text"         "color": {
        "text": "var(--wp--preset--color--da": ""h5": {
      "color": {
        "text"    "color": {
        "text": "var(--wp--preset--color--dark-b": ""h6": {
      "color": {
        "text": "var(--wp--preset--color--dark-blue)"
      }
    }
  },
```

Individual blocks, whether part of the core package, or added by a user, can provide their own styles. These are different than the styles provided on each individual block configuration.

Where they use custom properties, these properties also use the `--` separator.

```
  "blocks": {
```

```
      "core/post-title": {
        "typography": {
          "fontSize": "var(--wp--preset--font-size--x-large)"
        }
      },
      "var(--wp--preset--font-size--x-large)""core/paragraph"fontSize": "
          "fontSize": "var(--wp--preset--font-size--medium)"
        }
      },
      "core/post-date"Size": "var(": ""typography": {
          "fontSize": "var(--wp--preset--font-size--small)"
        }
      },
    }
  },
```

We can also define the template parts that we want to make available for theme development by adding them to the `templateParts` section. Where we know in advance (like headers and footers, we can specify the area as well).

```
  "templateParts": [
    {
      "name": "footer",
      "footer""title"r",
      "area": "foote": "   },
    {
      "name": "hea": ""name""name"le": "Header",
   ": ""title""header"
    },
  "header""area": "header"
    },
  ],
```

We can also define custom templates that we will use on our theme. We can also specify the types of content the template applies to.

```
  "customTemplates": [
    {
      "name": "page-sidebar-left",
      "page-sidebar-left""title": "Two columns, left sidebar",
      "postTypes"      ]
    },
    {
      "name": "page-templ": ""name"ns",
      "title": "Template for ": ""title"rn layouts",
      "postTypes": [
        "page",
       "page": [
        "page",
        "post"
      ]
    }
  ]
}
```

With the code we've discussed in this post we have the basis of a working theme. We can edit the theme in the Full Site Editor or creating the templates manually.

Creating templates manually

When working with Gutenberg template parts and templates we have two options:

- Create them in the full site editor
- Create them manually using the appropriate markup

This post will cover the later option and will serve as an overview of the markup we need to use to create the templates.

Gutenberg template markup is written inside HTML comment tags. This example shows a basic Gutenberg element and its attributes:

```
<!-- wp:query-title {"type":"archive"} /-->
```

The components of the element are:

- The wordpress rivendellweb, `wp:`
- The name of the element, `query-title`
- And any attributes as value pairs inside curly brackets, `{"type":"archive}`

The attributes are dependent on the element and not all attributes apply to all the available elements.

Building template parts

The next example shows a template part built around a post query loop, a replacement for the traditional PHP loop.

Breaking up the template in smaller sections to walk through them.

`wp:query` runs a query against the WordPress database and returns the results as specified in the query parameters.

```
<!-- wp:query-title {"type":"archive"} /-->
<!-- wp:term-description /-->
<!-- wp:query {
  "queryId":1,
  "query":{
    "pages":"100",
    "offset":0,
    "postType":"post",
    "categoryIds":[],
    "tagIds":[],
    "order":"desc",
    "orderBy":"date",
    "author":"",
    "search":"",
    "sticky":"",
    "exclude":[],
    "perPage":5,
    "inherit":true
    }
  } -->
```

After defining the query, we build the markup for the content of the query.

```
<div class="wp-block-query">
<!-- wp:post-template -->
<!-- wp:post-title {"isLink":true} /-->
<!-- wp:post-featured-image /-->
<!-- wp:group {"className":"is-style-valinor-box-shadow post-meta","backgr
<div class="wp-block-group is-style-valinor-box-shadow post-meta has-light
  <div class="wp-block-group is-style-valinor-box-shadow post-meta has-lig
  <!-- wp:post-author /-->
  <!-- wp:post-terms {"term":"category"} /-->
  <!-- wp:post-terms {"term":"post_tag"} /-->
</div><!-- /wp:group -->
```

Most of the templates deal with the post content.

`wp:title` displays the title of the block. Using the `isLink` attribute wraps the

title generates a link to the post.

`wp-post-featured-image` shows the post featured image if one is available.

`wp:spacer` generates vertical space between blocks. The `height` parameter indicates how big the separator is.

The child of the `wp:spacer` element is a `div` element with an inline style attribute matching the specified height in pixels.

The `div` element also has an `aria-hidden` attribute set to `true` so the document's accessibility tree will ignore it.

The `className` parameter is optional and, if used it must match one or more class names inside the child element's `style` attribute.

`wp:post-date` shows the data of the post was created.

`wp:post-author` contains the post author.

We then use the `wp:post-terms` elements twice: one to show the categories assigned to the post and the second one to show the tags assigned to the post (if any).

```
<!-- wp:spacer {"height":30} -->
<div style="height:30px" aria-hi<div style="height:30px" aria-hidden="true
```

We add another spacer before we add the content.

```
<!-- wp:post-excerpt {"moreText":"Continue reading"} /-->
<!-- wp:spacer {"height":20} -->
<div style="height:20px" aria-hidden="true" class="wp-block-spacer"></div>
<div style="height:20px" aria-hidden="true" class="wp-block-spacer"><!-- /
<!-- wp:separator {"color":"beige","className":"is-style-wide"} -->ge-back
<!-- /wp:separator -->
<!-- wp:spacer {"height":20} -->
<div style="<!-- wp:spacer {"height":20} --><!-- /wp:separator -->
<!-- wp:spacer {"height":20} -->
<div style="height:20px" aria-hidden="true" class="wp-block-spacer"></div>
```

```
<!-- /wp:spacer -->
<!-- /wp:post-template -->
```

`wp:excerpt` generates an except for the post or page. The `moreText` attribute provides the text for the link to read the rest of the content.

```
<!-- wp:query-pagination -->
<div class="wp-block-query-p<div class="wp-block-query-pagination"><!-- wp
</div>
<!-- /wp:query-pagination --></div>
<!-- /wp:query -->
```

The final piece of the `wp:query` element is the `wp:query-pagination` element that will generate the pagination links for the post.

Building templates

Building templates is simpler than building the parts for them. The next example shows a template to generate a page.

This template will use four template parts to generate a page.

We use `wp:template-part` to add the header, main and footer parts. We use a `wp:spacer` element to create a gap between the main and footer parts.

For each `wp:template-part` element we use, the `slug` attribute indicates the name of the part we want to use, the `tagName` attribute indicates the HTML tag we want to use to wrap the content we generate and `className` indicates the class name we want to add to the wrapping element.

```
<!-- wp:template-part {
  "slug":"header",
  "tagName":"header",
  "align":"full",
  "className":"site-header"
} /-->
```

```
<!-- wp:template-part {
  "slug":"main",
  "tagName":"main",
  "classNam":"site-main",
  "layout":{
    "inherit":true
    }
  } /-->
<!-- wp:spacer {"height":30} -->
<div style="height:30px"
  aria-hidden="true"
  class="wp-block-spacer"></div>
<div style="height:30px"
  aria-hidden="true"
  class="wp-block-spacer"><!-- /wp:spacer -->
<!-- wp:template-part {
  "slug":"footer",
  "tagName":"footer",
  "align":"full",
  "className":"site-footer"
} /-->
```

Creating parts and templates by hand is not easy and it’s not recommended. If you want to leverage the full site editor you should create your blocks and templates visually with the tools provided in the editor.

But sooner or later you will see the code for the template and it helps to understand how it works.

Locking blocks and templates

One of the things that has made me cautios about Gutenberg is how to give the client a set of plugins and themes that won't become a footgun. Yes, they own the product but that doesn't mean they will know how to use it properly.

WordPress provides two ways to lock templates, one from PHP and another one, less powerful, from Javascript/React.

The PHP solution requires using the `template_lock` property when registering the post type. This example provides a place holder for the `paragraph` block and then locks it

```php
<?php
function myplugin_register_template() {
    $post_type_object = get_post_type_object( 'post' );
    $post_type_object->template = array(
        array( 'core/paragraph',
          array(
            'placeholder' => 'Add Description...',
          )
        ),
    );
    $post_type_object->template_lock = 'all';
}

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

The possible values for `template_lock` are:

- **all** — prevents all operations. It is not possible to insert new blocks, move existing blocks, or delete blocks.
- **insert** — prevents inserting or removing blocks, but allows moving existing

blocks.

The value of `template_lock` is inherited by InnerBlocks.

If `templateLock` is not set in an InnerBlocks area, the locking of the parent InnerBlocks area is used.

If the block is a top level block, the locking configuration of the current post type is used.

Individual block locking

We can also choose to lock individual blocks. Block-level lock takes priority over the templateLock feature. Currently, you can lock moving and removing blocks.

Block-level locking is an experimental feature that may be removed or changed at anytime.

```
attributes: {
  // Prevent a block from being moved or removed.
  lock: {
    remove: true,
    move: true,
  }
}
```

The possible values for lock at the block level are:

- **remove** — prevents users from removing the block
- **move** — stops users from moving the block

Block locking takes precendence over template locking. You can customize the locking behavior using `templateLock` to lock all templates and blocks and then override it in individual blocks as needed. This is one way to prevent users from modifying the themes and templates after we've published them.

Styling blocks

Block Styles allow alternative styles to be applied to existing blocks whether they are defined in core or via third party plugins. They add a `className` attribute matching the style name to the block's wrapper.THe attribute applies alternative styles for the block if the style is selected.

The example registers a *narrow-paragraph* style for the core/paragraph block. When the user selects the narrow-paragraph style from the styles selector, an `is-style-narrow-paragraph` className attribute will be added to the block's wrapper.

```
wp.blocks.registerBlockStyle(
  'core/paragraph', {
    "name": "narrow-paragraph",
    "label": "Narrow Paragraph",
  }
)
```

By adding `isDefault: true` you can mark the registered block style as the one that is recognized as active when no custom class name is provided. It also means that there will be no custom class name added to the HTML output for the style that is marked as default.

To remove a block style use `wp.blocks.unregisterBlockStyle()`.

```
wp.blocks.unregisterBlockStyle(
  'core/quote',
  'large'
);
```

The above removes the block style named large from the core/quote block.

When unregistering a block style, there can be a race condition on which code runs first: registering the style, or unregistering the style. You want your unregister code to run last.

Do this by specifyin the component that is registering the style as a

dependency, in this case wp-edit-post. Additionally, using `wp.domReady()` ensures the unregister code runs once the dom is loaded.**

Enqueue your JavaScript with the following PHP code:

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

The JavaScript code in `rivendellweb-blocks.js`:

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

Server-side registration helpers

While the samples provided do allow full control of block styles, they do require a considerable amount of code.

To simplify the process of registering and unregistering block styles, two server-side functions are available: `register_block_style`, and `unregister_block_style`.

register_block_style

The `register_block_style` function receives the name of the block as the first argument and an array describing properties of the style as the second argument.

The properties of the style array must include name and label:

- **name**: The identifier of the style used to compute a CSS class.
- **label**: A human-readable label for the style.

Besides the two mandatory properties, the styles properties array should also include an inline_style or a style_handle property:

- **inline_style**: Contains inline CSS code that registers the CSS class required for the style
- **style_handle**: Contains the handle to an already registered style that should be enqueued in places where block styles are needed

It is also possible to set the `is_default` property to true to mark one of the block styles as the default one.

The following code sample registers a style for the quote block named “Blue Quote”, and provides an inline style that makes quote blocks with the `Blue Quote` style have blue color:

```
<?php
register_block_style(
    'core/quote',
    array(
        'name'         => 'blue-quote',
        'label'        => __( 'Blue Quote', 'textdomain' ),
        'inline_style' => '.wp-block-quote.is-style-blue-quote { color: b
    )
);
```

Alternatively, you can register a stylesheet that contains all your block-related styles and just pass the stylesheet’s handle so `register_block_style` function will make sure it is enqueue.

```
<?php
wp_register_style( 'myguten-style', get_template_directory_uri() . '/custo

register_block_style(
    'core/quote',
    array(
```

```
        'name'         => 'fancy-quote',
        'label'        => __( 'Fancy Quote', 'textdomain' ),
        'style_handle' => 'rivendellweb-block-styles',
    )
);
```

unregister_block_style

`unregister_block_style` allows developers to unregister a block style previously registered **on the server** using `register_block_style`.

The function has two arguments:

- The registered name of the block
- The name of the style

The following example unregisters the style named `fancy-quote` from the quote block:

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

The function unregister_block_style will 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.

We'll revisit styling blocks when we talk about Gutenberg as a design system.

Nesting blocks

Often we want to create a block with multiple child blocks. For example, block quotes may have lists and other elements nested inside them.

Container blocks like the columns blocks also support templates. This is achieved by assigning a nested template to the block.

This PHP example creates a block with two items, one of them with additional nested children:

- A root level paragraph with a placeholder
- A columns block that will contain one or more children
- A column
    - An image
- A column
    - A paragraph

```
<?php
$template = array(
  array( 'core/paragraph', array(
    'placeholder' => 'Add a root-level paragraph',
  ) ),
  array( 'core/columns', array(), array(
    array( 'core/column', array(), array(
        array( 'core/image', array() ),
    ) ),
    array( 'core/column', array(), array(
      array( 'core/paragraph', array(
        'placeholder' => 'Add a inner paragraph'
      ) ),
    ) ),
  ) )
);
```

We can also use Javascript / JSX block will create a blog and allow for child blocks to be added to it with the `InnerBlocks` property.

It also shows how we can constrain the content of the block children to a

specific list; In this example, we only allow paragraph and images to the demo-01 block.

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

const ALLOWED_BLOCKS = [
'@wordpress/block-editor'aragraph'
];

registerBlockType( 'rivendellweb-blocks/demo-01', {

  edit: () => {
    const blockProps = useBlockProps();

    return (
      'rivendellweb-blocks/demo-01'<div { ...blockProps }>
          <InnerBlocks allowedBlocks={ ALLOWED_BLOCKS } />;
      </div>
    );
  },

  save: () => {
    const blockProps = useBlockProps.save();

    return (
      <div { ...blockProps }>
          <InnerBlocks.Content />
      </div>
    );
  },
} );
```

One of the earliest problems I had with Gutenberg was that I couldn't create a blockquote if the text had lists in it. The issue appears to have been fixed but having this as a backup is always a good idea eventhough it requires more work to implement.

@wordpress/create-block

`@wordpress/create-block` (`create-block` for short) is a Node package that provides the officially supported way to create blocks.

The basic structure the script provides is as follows:

```
.
├── build
|   ├── block.json
|   ├── index.asset.php
|   ├── index.css
|   ├── index.css.map
|   ├── index.js
|   ├── index.js.map
|   ├── style-index.css
|   └── style-index.css.map
├── node_modules
├── demo-block.php
├── package-lock.json
├── package.json
├── readme.txt
└── src
    ├── block.json
    ├── edit.js
    ├── editor.scss
    ├── index.js
    ├── save.js
    └── style.scss
```

The script also installs the `@wordpress/scripts` package, which is a set of tools to make working with WordPress easier.

You can run the package with the following command inside your server's **plugins** directory:

```
npx @wordpress/create-block rw-demo-block \
```

```
    --template @wordpress/create-block-tutorial-template \
    --title "Demo Block" \
    --namespace rivendellweb \
    --short-description "Another awesome block" \
    --category "widgets"
```

This command will give you all the tools you need to create a block along with a pre-filled plugin to start working from.

The command **will not** install the plugin on your server. You have to do the installation and activation manually.

You can also create your own base templates and use them with `create-block` and host them in Github so you can run them with the `create-block` command.

The template contains at minimum the following files:

- An `index.js` file to run the template creation
- One or more `mustache` templates that will be processed when we create a block based on the template. These templates include scss style
- Optional `package.json` with the template metadata. We are not installing any modules from the template.

Once you have the template ready to go you can run the code locally

```
npx @wordpress/create-block --template ~/path/to/my-template/
```

or from NPM

```
npx @wordpress/create-block --template my-template
```

For a good example of how custom templates work, see Marcus Kazmierczak's overview of the process in Make your own create-block templates. You can also see the code from his tutorial on Github and use it on your projects by running:

```
npx  @wordpress/create-block --template mkaz-block-template
```

The `create-block` script automates a lot of the tedious tasks of creating a block. There are other ways to do it, but this is the way the WordPress team recommends.

Revisiting Gutenberg as a design system

Gutenberg presents an interesting way to create and use design systems and present them to the user.

I use the following definition of a design system:

> A design system is a complete set of standards intended to manage design at scale using reusable components and patterns.
>
> Design Systems 101 — Nielsen Norman Group

I wrote a series of posts on the topic of using Gutenberg as a design system and variations of those design items.

- Gutenberg as a design system
  - Part 1
  - Part 2
  - Part 3

This post is a review and further exploration of Gutenberg-based design systems. This post will borrow some concepts from Brad Frost's atomic design.

Below is Brad's Presentation about Atomic Design, given at Beyond Tellerand in 2013

Start from the beginning: Building Atoms

Atoms are the smallest building blocks of an Atomic Design System. In Gutenberg, atoms are the individual components and variations thereof.

The default blocks are well documented so we won't discuss them here.

Defining a color palette

One of the first things I look at is how to define a color palette. I will use the different palettes show in my full color palette created for a diffrent design system experiment.

Taking advantage of the `theme.json` file, we can define a color palette based on the list of colors available to use.

Usually the client will give you a list of colors that match their brand or you can create your own list.

Inside your `theme.json` file you can define your own colors inside the `palette` section. Each color has three attributes: `slug`, `color` and `name`.

```
"palette": [
  {
    "slug": "plum-crazy-purple",
    "plum-crazy-purple""color"ame": "Plum Crazy": ""name" },
  {
    "slug": "rebecca-purple": ""slug"lor": "#663399",
    "na": ""color": "#663399",
    "name" "slug": "butterscotch",
    "co": ""slug"8827",
    "name": "Bu": ""color": "#D48827",
    "name"cherry",
    "color": "#D2042D": ""slug"me": "Cherry"
  ": ""color""slug": "black-ch": ""name" "color": "#52253A",
   ": ""slug": "black-cherry",
    "color": "#52253A",
    "name": "Black Cherry"
  }
]
```

Because of the way colors are displayed in the editor, we need to be mindful of the number of colors we use on our themes.

We also need to be careful with color contrast between foreground and background colors. The last thing we want is for our text to be illegible because of poor cotrast with the background color.

Loading web fonts

One of the most contentious things in working with WordPress overall is how to use web fonts.

Ideally, this would be a matter of using fonts from CDNs like Google Fonts or Adobe Fonts but it's not so easy... we need to be mindful of the following:

- German Court Rules Websites Embedding Google Fonts Violates GDPR
- Website fined by German court for leaking visitor's IP address via Google Fonts

So we might have to use locally hosted web fonts or use system fonts.

Since I'm trying to recreate an existing theme as an example, I will try to use the same local web font that I used in the original.

Recursive is a variable font that I've been working with since it was in beta. It provides pretty much everything that you may need:

- A monospace code axis for pre-formated and code blocks
- Sans-serif, and casual axes for typographical work
- Variable weight axis from light to extra black (300 to 1000 in numeric values)
- Slant and cursive axes to control italics behavior
- A set of presets that combine the different values from the available axes to give you control without having to drop to the low level value adjustment
- Support for 200 latin languages out of the box

TO get started we add the `@font-face` declaration to our stylesheet.

Variable fonts make some changes to the `@font-face` syntax we are used to.

`font-weight` now takes two values representing the lowest and highest boundaries for the attribute. In this case the font support weights between 300 and 1000.

I am using a subset of the full Recursive font that only includes lating languages. The `unicode-range` attribute indicates the specific Unicode codepoints that the font subset supports.

```
@font-face {
  font-family: "Recursive"Recursive"url("path/to/Recursive.woff2");
  font-weight: 300 1000;
  unicode-range: U+000D, U+0020-007E, U+00A0-00FF, U+0131, U+0152-0153,
      U+02BB-02BC, U+02C6, U+02DA, U+02DC, U+2007-200B, U+2010, U+2012-201
      U+2018-201A, U+201C-201E, U+2020-2022, U+2026, U+2030, U+2032-2033,
      U+2039-203A, U+203E, U+2044, U+2052, U+2074, U+20AC, U+2122, U+2191
      U+2193, U+2212, U+2215;
}
```

Next, we add the following default values to the `:root` element. We add them as CSS custom properties / variables so we can override themm later without having to worry about specificity later.

```
:root {
  --mono: "MONO" 0;
  --casl: "CASL" 0;
  --wght: "wght" 400;
  --slnt: "slnt" 0;
  --crsv: "CRSV" 0.5;
}
```

We can then apply the properties to the `html` or body elements to set the default.

This command will set the default font for the document (the body element and all its children) to Recursive with a system sans-serif backup. It will also change the parameters of the font to what we specify or to the values in the `:root`

```
body {
  font-family: "Recursive", sans-serif;
  font-variation-settings:
    var(--mono),
    var(--casl),
    var(--wght),
    var(--slnt),
    var(--crsv);
```

Using CSS variables allow us to override the default values by reassigning them.

In the following example we change the value of the casual axis from 0 (normal) to 1 (casual). This will make all h2 elements appear in a more casual and playful font.

```
h2 {
  --casl: "CASL" 1;

  font-variation-settings:
    var(--mono),
    var(--casl),
    var(--wght),
    var(--slnt),
```

```
    var(--crsv);
}
```

The axes described in this post only work with the Recursive variable font.

Using the `font-variation-settings` syntax is not recommended. However, support for the recommended `font-variant-*` properties is uneven and depends on the feature we're using

Block variations

Another way we can create our atoms and other design elements is to use block variations.

The variations API allows the creation of new variants of a block that share a common structure. This reduces the potential number of blocks we need to create and provides for a more consistent design system.

The following example shows how to create block variations **for default blocks** and package them as a plugin. The same thing can be done in a theme if we want to tie the variations to a specific theme.

The first step is to create the PHP file that will make the plugin work. Without the comment, the package will not be recognized as a plugin.

We then make sure that the plugin cannot be accessed directly.

The core of the PHP file are the functions that enqueue the Javascript file containing our variations and the stylesheet that implements the variations' CSS styles.

`rivendellweb_enqueue_variations` enqueues and loads the script that holds the variations of the core blocks.

`rivendellweb_variation_styles` enqueues the styles associated with the variations.

```
<?php
/**
```

```
 * Plugin Name: Rivendellweb variations
 * Plugin URI: https://github.com/caraya/rivendellweb-variations
 * Description: Rivendellweb blocks variations.
 * Version: 0.1.0
 * 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-po'ABSPATH'      file
    );
}
add_action( 'enqueue_block_editor_assets', 'rivendellweb_enqueue_variation

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' );
```

In `block-variations.js` we defined the variations we want to create. We use regusterBlockStyle to define these variations.

> `registerBlockStyle` has three basic parameters:

1. The name of the block we are creating the variation for

2. The computer-readable name used to compute the class name of the variation
3. The human readable label

```
wp.blocks.registerBlockStyle( 'core/paragraph', {
  name: 'lede-paragraph',
  label: 'Lede (first) Parag'lede-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',
})
```

THe final piece is the CSS necessary to style the variations.

The classes use the following syntax: **.is-style-{class-name}**

```
.is-style-lede-paragraph {
  font-size: 1.5em !important;
}

.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: 0;
```

```
}

.is-style-red-quote {
  border-left: 0;
  color: red;
  font-size: 5vmax;
}
```

So far the variations work, but they require a lot of code and three different files to be susscesful.

Registering the variations on the server reduces the amount of complexity we need to maintain.

The idea is to use PHP's register_block_style function to consolidate all the information for the variations into one file.

The function takes two parameters:

1. The name of the block we are creating the variations for
2. An array of properties for the style
    1. **Name**: The name of the variation used to compute the class name (**required**)
    2. **Label**: The human-readable label (**required**)
    3. inline_style: Contains inline CSS code that registers the CSS class required for the style (**optional**)
    4. **style_handle**: Contains the handle to an already registered style that should be enqueued in places where block styles are needed (**optional**)
    5. **is_default**: Boolean value. If set to true it indicates that this variation is the default variation for the block (**optional**)

This example adds a blue quote varation to the `core/quote` block.

```
<?php
register_block_style(
    'core/quote',
    array(
```

```
        'name'         => 'blue-quote',
        'label'        => __( 'Blue Quote', 'textdomain' ),
        'inline_style' => '.wp-block-quote.is-style-blue-quote { color: b
    )
);
```

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

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

The following example unregister the blue quote variation.

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

Important

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

Block Patterns: Molecules, Oragnisms and Templates

In the Gutenberg world Block Patterns are predefined block layouts, available from the patterns tab of the block inserter.

Register block categories

Before defining the block patterns I'll make available via the plugin, I like to define one or more categories to collect the patterns under.

Just like with blocks, the categories help organize the content in our own categories. A block or pattern can belong to more than one category.

```
<?php
function rw_custom_register_my_pattern_categories() {
  register_block_pattern_category(
      'rivendellweb',
      array( 'label' => __( 'Rivendellweb', 'rw-custom' ) )
  );
}

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

Registering block patterns

Once we install them we can then further customize and expand them

The register_block_pattern helper function receives two arguments.

- **title**: A machine-readable title with a naming convention of namespace/title
- **properties**: An array describing properties of the pattern.

The properties available for block patterns are:

- **title**: A human-readable title for the pattern (**required**)
- **content**: Block HTML Markup for the pattern (**required**)
- **description**: A visually hidden text used to describe the pattern in the inserter (**optional**) categories: An array of registered pattern categories used to group block patterns. Categories are registered using `register_block_pattern_category` (**optional**)
- **keywords**: An array of aliases or keywords that help users discover the pattern while searching (**optional**)
- **viewportWidth**: An integer specifying the intended width of the pattern to allow for a scaled preview of the pattern in the inserter (**optional**)

Make sure that you add the blocks in the init hook as shown in the example.

```
<?php
register_block_pattern(
  'rivendellweb-patterns/demo-pattern',
  array(
    'title'       => __( 'Two buttons', 'my-plugin' ),
    'description' => _x( 'Two horizontal buttons, the left button is fille
    'content'     => "<!-- wp:buttons {\"align\":\"center\"} -->\n<div cl
  )
);

add_action( 'init', 'my_plugin_register_my_patterns' );
"</a></div>\n<!-- /wp:button --></div>\n<!-- /wp:buttons -->"'Button Two'
  )
);

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

The syntax for the pattern context takes a while to learn and it's not intuitive. The best way to learn the syntax is to look at existing patterns that match what you want to do and use them as a starting point. You can build the pattern in the Gutenberg Editor and then copy it into `content` parameter of the pattern you want to create.

Because we're working with PHP, we need to escape the HTML we use in the patterns to prevent misuse.

For each registration method there is a corresponding unregistration method.

Further extensions: Custom Post Types

Another way to create elements for a design system is to use custom post types

We can use them with limited templates and patterns or we can createae custom patterns for the CPTs or we can use CPTs as an authoring tool for content templates.

Content templates

Using Block Patterns as content templates an interesting way to create content patterns.

These patterns can be as sinmple as the following example that describes an incident report.

```
<!-- wp:heading -->
<h2>Summary</h2>
<h2><!-- /wp:heading -->

<!-- wp:paragraph -->What happened?</p>
<!--</p><!-- /wp:paragraph -->

<!-- wp:heading -->eline</h2>
<!-- /wp</h2><!-- /wp:heading -->

<!-- wp:paragraph -->it happen?</p>
<!-- /wp:para</p><!-- /wp:paragraph -->

<!-- wp:heading -->
<h2>Impact</h2>
<!-- /wp:heading -->

<!-- wp:paragraph -->ted?</p>
<!-- /wp:paragrap</p><!-- /wp:paragraph -->
<h2>Process</h2>
<!-- /wp:heading -->
<h2><!-- /wp:heading -->

<!-- wp:paragraph -->/p>
<!-- /wp:paragraph -->

<!-- wp:heading -->
<h2>I<!-- wp:heading --><!-- wp:heading --><!-- wp:paragraph -->
<!-- wp:paragraph --><!-- /wp:heading -->
```

```
<!-- wp:paragraph -->
<p>What we are doing to prevent happening again</p>
<!-- /wp:paragraph -->
```

If we write this pattern in the Gutenberg Editor then we get a visual experience of the pattern we're creating and a visual approximation to what it will look like when the pattern is used.

The article also presents a tool to create patterns from posts of a custom post type.

More information

- WordPress CLI
    - wp scaffold post-type
- The Traditional Way
    - Registering Custom Post Types
- The Gutenberg Way
    - How to Use Gutenberg with WordPress Custom Post Types
    - Using Block Patterns as content templates
    - How to Build Block Patterns for the WordPress Block Editor

Pagination and Post Navigation

Gutenberg gives you the tools to do pagination visually but, unfortunately, I haven't found a way to customize the pagination tool so that it works the way I want to.

There are two different types of pagination in a WordPress-based blog: One for index and archive pages and one for individual posts.

Index Page Navigation

THe pagination layout I ended up using on my site looks like this:

```
<!-- wp:query-pagination {
  "layout":{"type":"flex" "justifyContent":"center"}} -->
  <!-- wp:query-pagination-previous /-->

  <!-- wp:query-pagination-numbers /-->

  <!-- wp:query-pagination-next /-->
<!-- /wp:query-pagination -->
```

There are some issues with the default navigation.

- If there is no previous or next post, the remaining content will move. I would rather is remain static and the space be left empty so the navigation remains centered
- Because the page navigation is built inside the query-loop block, and we've styles the loop to be narrower so it looks acceptable the navigation is also constrained by the styles on the loop

I've made the concious decision to let the previous and next post links overflow the parent element. However when both previous and next links are present the links are squished together and it doesn't look like the design I intended.

As a hacky workaround, for now, I've added this CSS to the site's

customizations:

```
.wp-block-query-pagination {
    width: 80vw
    margin-left: -225px;
}
```

The negative margin will move the pagination container to the left to try and match the width of the footer below.

I also need to test in other desktop and mobile devices to make sure that the pagination still works as intended there.

Post Navigation

Post navigation is different than the page navigation and it uses a lot more attributes in the Gutenberg comments.

Structurally, the layout is built around a two column layout and leverages a lot of built-in block attributes and parameters.

Each `post-navigation-link` element has three attributes that are relevant:

- `type`: This is the type of link. It can be `previous` or `next`
- `showTitle`: This is a boolean that determines if the title of the post should be shown
- `linkLabel`: Boolean that indicates if the link should have a label

```
<!-- wp:columns {"style":{"spacing":{"margin":{"top":"2.38rem","bottom":"2
  <div class="wp-block-columns post-nav" style="margin-top:2.38rem;margin-
    <div class="wp-block-columns post-nav" style="margin-top:2.38rem;margi
    <div class="wp-block-column is-vertically-aligned-center">
      <!-- wp:post-navigation-link {"type":"previous","showTitle":true,"li
    </div>
    <!-- /wp:column -->

    <!-- wp:column -->
    <div class="wp-block-column is-style-default">
```

```
      <!-- wp:post-navigation-link {"textAlign":"left","showTitle":true,"
    </div>
    <!-- /wp:column -->
  </div>
<!-- /wp:columns -->
```

I like that the navigation links have attributes matching functionality, I just wish they were better documented for those of us wishing that we could edit them directly on the template.

Gutenberg: My Issues (2022 version)

This is a place to document issues I'm finding as I build my theme. A lot of these issues may be cause by my lack of experience with Gutenberg and the way the full site editor works.

I will file issues about these items in the Gutenberg repo where I feel it's necessary.

Adjusting Expectations

One of the things I'm not sure works as intended or that I'm not understanding correctly is how the full site editor works.

If I'm understanding this correctly, the changes on the editor should be reflected to the corresponding file I'm editing and viceversa, right?

This is not the case. I've made extensive changes to the theme I'm working on and none of those changes are reflected in the file I'm working with. Likewise, I've made significant changes in template files and they are not reflected in the editor.

I asked the question in the WordPress Slack and the answer I got was it was working as designed.

> That's actually deliberate, even if it's initially confusing. The customizations that you use within the site editor are saved as CPTs. You can export the templates to get the HTML version if you want to put them into the code, but this way the user's changes are non-destructive and you can safely update the theme or even change themes and keep your templates. You can't export the theme.json file yet, but that's coming. (edited)
>
> From: Make WordPress Slack — core-editor discussion

It appears that if I want to continue creating block themes, I will have to use the full site editor as the source of truth and remember to export the files periodically and to update the editor to match the layout if I'm making changes directly to the

template.

See these Github Issues for context and discussion on the isue:

- How to sync edits made within a theme's HTML files to the Block Editor (Full-site editing)
- See this comment in particular for a way to potentially edit content on the file system and have it automatically reflect on the editor and the front end
- On Locking and TemplateLocking.

Poor or non-existant block documentation

Documentation for the blocks API is sparse and, almost, non existant.

I thought the Storybook section of the Gutenberg repo may help but I found out that it is for Gutenberg's low-level components, not the blocks that we use in the editor.

I believe the only way to get a good idea of what the core blocks can do is to create a layout that you like and then copy it or export it into your favorite editor to see how it works and if there are any attributes worth documenting for future use.

Mixing HTML with template content

I'm trying to get something like this in the full site editor:

```
<p>Posted by: <a href="link/to/author/archive">Carlos</a></p>
<p>Posted on: <a href="link/to/date/archive">February 22, 2021</a></p>
```

The current layout is a nested column layout. The label is a paragraph element, and the author name and date posted are Gutenberg elements inside a `div` element so we can make it into a flexbox layout.

```
<!-- wp:column {"width":"100%"} -->
<div class="wp-block-column" style=<div class="wp-block-column" style="fle
```

```
    <!-- wp:column {"width":"50%"} -->
    <div class="wp-block-column" style="flex-basis:50%">
      <!-- wp:paragraph {"fontSize":"s<!-- wp:column {"width":"50%"} --><
        <p class="has-small-font-size">Posted by:</p>
      <!-- /wp:paragraph -->
      <!-- wp:post-author-name {"fontSize":"small"} /-->50%"} -->
    <div class="wp-block-column" style="flex-basis:50%">
      <!-- wp:paragraph {"fontSize":"small"} -->
      <p class="has-sma<div class="wp-block-column" style="flex-basis:50%
    </div>
    <!-- /wp:column </div><!-- /wp:paragraph -->
      <!-- wp:post-date {"isLink":true,"fontSize":"small"} /-->
    </div>
    <!-- /wp:column -->
  </div>
  <!-- /wp:columns -->
</div>
<!-- /wp:column -->
```

I can't seem to figure out if there is a pattern similar to what we do for `wp:post-navigation` links where there is a label available (even though we can't edit its content) inserted as a span that we can tweak with CSS.

Again, I'm not sure if this is something I'm doing wrong or if it's something that is just not available in Gutenberg at all.

Navigation issues

If there is no previous or next post, the remaining content will move. However when both previous and next links are present the links are squished together and it doesn't look like the design I intended, regardless of the code I use.

When both previous and next links are present the links break down into two lines and they only take the width of the query-loop block.

I came around with a CSS solution that kind of does what I need it to. The negative margin is a hack to make the navigation stay centered and match the footer below it.

```
div[class*="wp-container"].wp-block-query-pagination {
  width: 80vw;
  margin-left: -250px;
}
```

There is more research I need to do to make sure this won't break on desktop screens with different resolutions.

How many sources for blocks are enough?

In an ideal world, installing all the block libraries that we want wouldn't be a problem, right?

Unfortunately, installing multiple block collections can be problematic and may cause performance issues and confuse users when a given block breaks because the package it was installed from hasn't been installed in the new WordPress installation.

SO how do we plan for the number of block plugins that we install? Do we need to install external packages for added functionality?

Creating header navigation menus

Surprisingly one of the hardest part of building a theme was adding a navigation for the header element.

At the most basic, the `wp:navigation` has a single attribute that links the menu to the underlying navigation post using the post's ID attribute.

```
<!-- wp:navigation {"ref":295} /-->
```

The problem I've had is that I can't figure out how to build the navigation page so, until I figure out how to build the menu page, I've built a static menu with hardcoded links. The modified link looks like this:

```
<!-- wp:navigation {
  "className":"main-navigation",
  "layout":{
    "type":"flex",
    "setCascadingProperties":true,
    "justifyContent":"right",
    "alignItems":"center"
    }
  } -->

  <!-- wp:navigation-link {
    "label":"About",
    "title":"about",
    "type":"page",
    "url":"/about/"
  } /-->

  <!-- wp:navigation-link {
    "label":"Privacy",
    "title":"Privacy",
    "type":"page",
    url":"/privacy"
  } /-->

  <!-- wp:navigation-link {
    "label":"Codepen",
    "title":"Codepen",
    "type":"link",
    "url":"https://codepen.io/caraya"
  } /-->

  <!-- wp:navigation-link {
    "label":"Projects",
    "title":"Projects",
    "type":"link",
    "url":"https://caraya.github.io/mprojects/"
  } /-->
```

```
  <!-- wp:navigation-link {
    "label":"Patterns",
    "title":"Patterns",
    "type":"link",
    "url":"https://rivendellweb-patterns.netlify.app/"
  } /-->

  <!-- wp:navigation-link {
    "label":"Layouts",
    "title":"Layouts",
    "type":"link",
    "url":"https://rivendellweb-layout-experiments.netlify.app/"
  } /-->
<!-- /wp:navigation -->
```

The information available about thenavigation block doesn't cover how to build the navigation pages. Until I do then I'll have to keep the static link around, perhaps as a separate pattern.