Gutenberg review: blocks

In a previous post I briefly discussed how to configure blocks and themes using json files. The emphasis of that post was on theme configuration and briefly mentioned `block.json` in passing as a way to configure individual blocks.

This post will dive deeper into `block.json` as a block configuration tool and how to use the configuration data in the code of the plugin we create.

```
npx @wordpress/create-block notice \
--namespace rivendellweb
```

The command will produce the following scaffolding in the `notice` directory. It will also install `@wordpress/scripts` as part of the installation process.

I've removed the content of `node_modules` for brevity.

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

The installation will also add the `block.json` file that we will be working with.

Some of the features of the build.json file that need a little explanation follow.

`$scehma` indicates the location of the WordPress block JSON schema. This schema makes it easier to write and validate the block configuration.

`apiVersion` is the version of the block configuration schema. Newer blocks should use the latest version available.

`title` and `category` indicate the title and category assigned to the block. These can be either built-in or custom values created on the backend using PHP.

`supports` indicates the features that the block supports.

Opting into any of these features will register additional attributes on the block and provide the UI to manipulate that attribute.

In order for the attribute to get applied to the block the generated properties get added to the wrapping element of the block. They get added to the object you get returned from the useBlockProps hook.

`textdomain` is the i18n text domain for the block.

`editorScript` and `editorStyle` are the scripts and stylesheets that get added to the block editor.

`style` is the stylesheet that gets added to the block in the frontend.

```
{
  "$schema": "https://sche"https://schemas.wp.org/trunk/block.json""apiVe
  "version": "0": ""version"le": "Notice"": ""title"ry": "widgets"": ""ca
  "icon"n": "Example b": ""description": "Example block written with ESNe
  "supports": {
    "html""notice",
  "ed"notice""textdomain": "notice",
  "editorScript" "file:./index.css",
  "file:./index.css""editorStyle": "file:./index.css",
  "style": "file:./style-index.css"
}
```

This is the basic block configuration for a single block inside a plugin. For more

information about settings available in the block api, see the Block API Reference.

Leveraging plugin templates

One of the most interesting features of `create-block` is the ability to use a template to generate the block.

When using the `--template` attribute we can point to a given Github repository containing a template formatted in a certain way.

```
npx @wordpress/create-block example-plugin \
--template @ryanwelcher/hello-block-template
```

Dynamic versus static blocks

There are two types of blocks available: Static and Dynamic. We'll cover them in detail below.

Static Blocks

Most of the blocks we use are static. They will output the content inside them and don't require any additional configuration.

This is what you'll see most of the time when working with Gutenberg.

To create a static plugin run the following command:

```
npx @wordpress/create-block example-plugin \
--namespace rivendellweb
```

Replace `example-plugin` with the name of your plugin and `rivendellweb` with a namespace for your plugin. The namespace should be unique as it will help prevent naming conflicts with other code in plugins or core. The namespace will also be used to generate the block name (namespace/block-name).

The `save` and `edit` functions are required. The edit function will be used to generate the block in the editor.

```
export default function Edit() {
  return (
    <p {...useBlockProps()}>
      {__('Demo Block – hello from the editor!', 'demo-block')}
    'demo-block'</p>
  );
}
```

The save function will be used to generate the block in the frontend.

```
export default function save() {
  return (
    <h2 {...useBlockProps.save()}>
      {__('Demo Block – hello from the saved content!', 'demo-block')}
    'demo-block'</h2>
  );
}
```

Code-wise, the main difference between the edit and save method is on what you return.

The edit function returns the destructured `useBlockProps` object.

The save function returns the destructured `useBlockProps.save` object. This is essentially the rendered content of the block.

Dynamic blocks

Dynamic blocks allow users to enter data to customize the block. Because of their dynamic nature, dynamic blocks need additional configuration beyond what we do for static blocks.

Rather than starting from scratch I'm using Ryan Welcher's dynamic block template. We use it with the following command:

```
npx @wordpress/create-block example-plugin \
--template @ryanwelcher/dynamic-block-template
```

Because of their nature, we need to do more work to make them work. The fist step is in the block registration: We add a `render_callback` property to the block configuration object with the name of the function that will execute to render the block.

```
<?php
function rivendellweb_demo_dynamic_plugin_block_init() {
  register_block_type(
    plugin_dir_path( __FILE__ ) . 'build',
    array(
      'render_callback' => 'rivendellweb_demo_dynamic_plugin_render_callba
    )
  );
  }
  add_action( 'init', 'rivendellweb_demo_dynamic_plugin_block_init' );
```

We then define the function we specified in the `render_callback` property.

We create a buffer using ob_start(), load the a template file usint require and then close the buffer using ob_get_clean().

```
<?php
function rivendellweb_demo_dynamic_plugin_render_callback( $atts, $content
  ob_start();
  require plugin_dir_path( __FILE__ ) . 'build/template.php';
  return ob_get_clean();
}
```

If use InnerBlocks in a dynamic block you will need to save the InnerBlocks in the save callback function using `<InnerBlocks.Content/>`.

The template can be as easy or as complex as we need to to be. For example, the code below will render our message using the appropriate css attributes and the i18n escaped text

```
<h2 <?php echo esc_attr( get_block_wrapper_attributes() ); ?>>
 <?php esc_html_e( 'Dyna'Dynamic Block Examples'mo-dynamic--plugin' ); ?>
```

```
</h2>
```

See Creating dynamic blocks for more information about dynamic blocks, and how to configure them.

Updating blocks: deprecations

There are times when we need to change the content of a block without breaking the block for the user.

A block can have several deprecated versions. A deprecation will be tried if the current state of a parsed block is invalid, or if the deprecation defines an `isEligible` function that returns true.

Deprecations do not operate as a chain of updates, like database migrations. Instead, this is how Gutenber handles deprecations:

- If the current save method does not produce a valid block, the first deprecation in the `deprecations` array is passed the original saved content
- If its save method produces valid content this deprecation is used to parse the block attributes
    - If it has a migrate method it will also be run using the attributes the deprecation parsed
- If the first deprecation's save method does not produce a valid block the subsequent deprecations in the array are tried until one Gutenberg encounters one that produces a valid block
- The attributes, and any innerBlocks, from the first deprecation to generate a valid block are passed to the current save method to generate new valid content for the block

At this point the current block should now be in a valid state and the deprecations workflow stops.

Note:

If a deprecation's save method does not produce a valid block then it is skipped

completely. If you have several deprecations for a block and want to perform a new migration, like moving content to `InnerBlocks`, you may need to update the migrate methods in multiple deprecations in order for the required changes to be applied to all previous versions of the block.

Note:

If a deprecation's save method imports additional functions from other files, changes to those files may accidentally change the behavior of the deprecation. You may want to add a snapshot copy of these functions to the deprecations file instead of importing them in order to avoid inadvertently breaking the deprecations.

For blocks with multiple deprecations, it is easier to save each deprecation to a constant with the version of the block it applies to, and then add each of these to the block's deprecated array. The deprecations in the array should be in reverse chronological order. This allows the block editor to attempt to apply the most recent and likely deprecations first, avoiding unnecessary and expensive processing.

```
const v1 = {
  // Deprecation data for version 1
};
const v2 = {
  // Deprecation data for version 2

};
const v3 = {
  // Deprecation data for version 3
};

// Apply the migrations in reverse order
const deprecated = [ v3, v2, v1 ];
```

To create the deprecations we need to do the followingn (in no particular order):

Create a file holding deprecation data for the block. We do this to make sure that the deprecations are in their own file are easier to read and don't make the block file more complicated than it needs to be.

```
import { useBlockProps } from '@wordpress/block-editor';
import { __ } from '@wordpress/i18n';

const v1 = {
  attributes,

  save() {
    return (
      <p {...useBlockProps.save()}>
        {__('Demo Block – hello from the saved content!', 'demo-block')}
      </p>
    )
  },
}

export default [v1]
```

Add a deprecated method to `registerBlockType`. This will help Gutenberg know that the block should be updated and what content to update from the old template.

We use Object Property Value Shorthand to assign values to the save and deprecated properties.

```
<?php
registerBlockType('create-block/demo-block', {

  edit: Edit,
  save,
  deprecated

})
```

Every time that we modify the markup of the block we need to create a new deprecation that will indicate the changes we made.