**Custom Gutenberg Block Plugin**

**Install Some Stuff**

● Create **package.json**:

● Install WordPress scripts

○ npm i @wordpress/scripts

● Update "scripts" in package.json

○ Copy from README

● Ignore directories in .gitingor

**Add Files & (Don't) Configure webpack & Babel**

● Create src/index.js

● Put some JavaScript in there.

● Compile it to make sure everything works.

○ npm run build ○ You should see built JavaScript

● Be impressed by how simple webpack and Babel are to use :)

● The one PHP file

**Write Block Now**

**Static, Non-editable Block**

● Open src/index.js

● Write Some JavaScript:

○ Import dependencies

○ registerBlockType()

○ Edit Callback

○ Save Callback

Transpile Is A Cool Word!

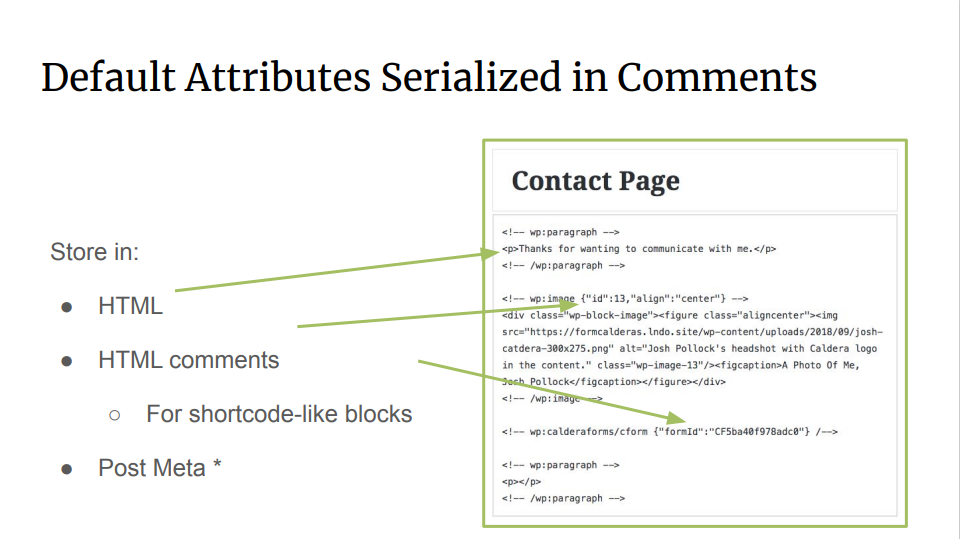
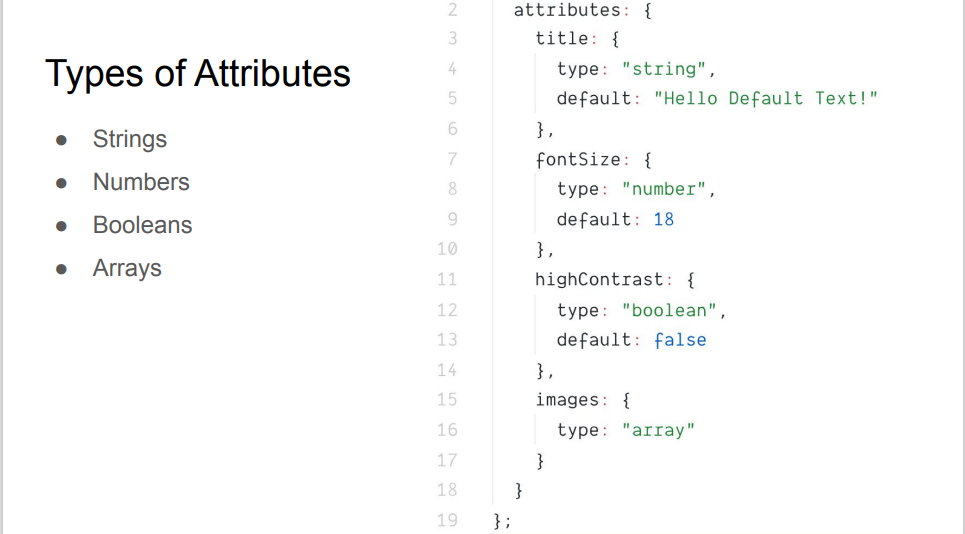
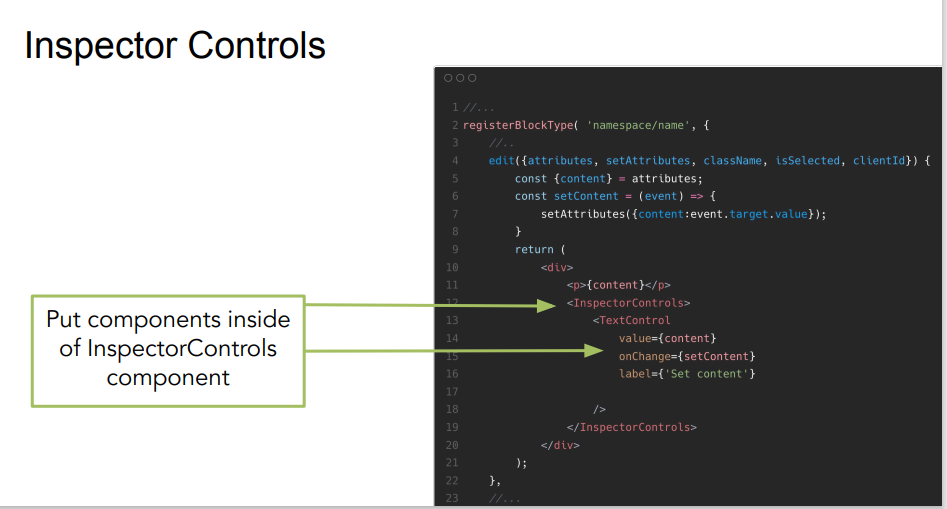
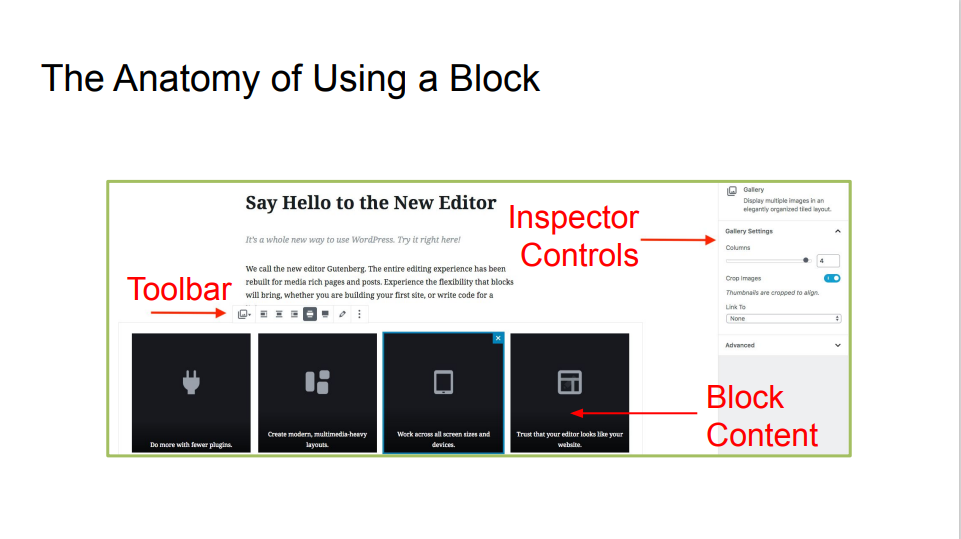
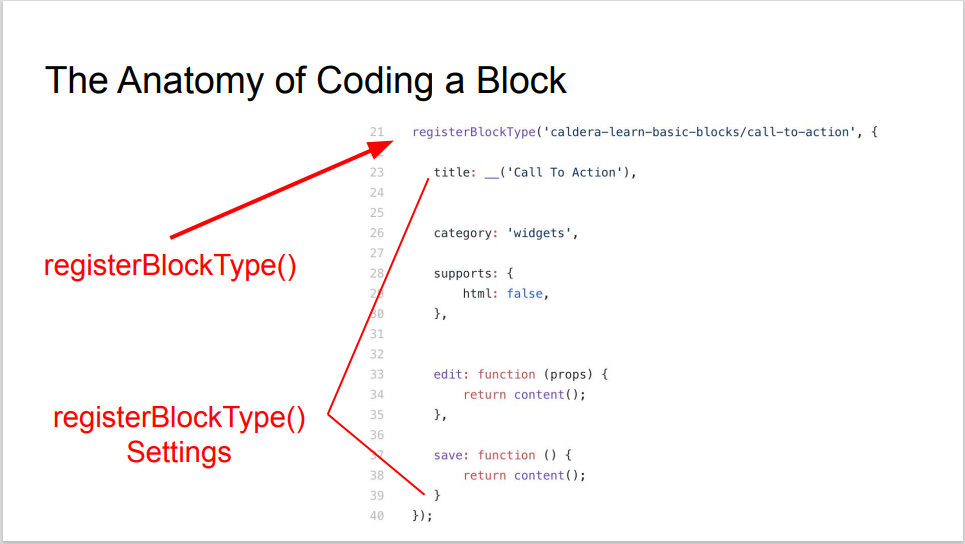
● For manual testing while developing

○ npm run start

● To ship for production

○ npm run build

**Editable Block**



**Attribute Storage Methods**

● Default - Serialized in HTML comments

● Sourced - Stored as part of the HTML content

● Post Meta - Stored in database as post meta

**Make Block Editable**

● Install new dependencies:

○ npm i @wordpress/editor @wordpress/components

● Import TextControl from @wordpress/components

○ Use for editting message

● Import InspectorControls from @wordpress/editor

**Actually Make It Editable**

● Add message attribute

● Get current value of message from attributes prop passed to save and edit.

● Create update function for message using setAttributes prop passed to edit callback.

● Display current value of message in edit and save callbacks.

● Use TextControl to created edit interface for block.

● Wrap the control in InspectorControls so it goes in the inspector controls.

**Taking Gutenberg Further**

● Shortcode to block conversion

● Plugin sidebars

● Block testing

● Isomorphic blocks

● React skills

● Redux Data Module

● Other frameworks besides React

**Syntax:**

register\_block\_type( *string|WP\_Block\_Type* $block\_type, *array* $args = array() ): [WP\_Block\_Type](https://developer.wordpress.org/reference/classes/wp_block_type/)|false

**Description:**

Registers a block type. The recommended way is to register a block type using the metadata stored in the block.json file.

The paths to compiled block files are defined in ./block.json. **The rest of the enqueuing process is handled for you automatically!**

{ ...

"editorScript": "file:./build/index.js",

"editorStyle": "file:./build/index.css",

"style": "file:./build/style-index.css"

}

## [Parameters](https://developer.wordpress.org/reference/functions/register_block_type/" \l "parameters)

$argsarrayOptional

* titlestring
* categorystring|null
* iconstring|null
* descriptionstring
* keywordsstring[]
* textdomainstring|null
* variationsarray
* render\_callbackcallable|null
* attributesarray|null

## [Return](https://developer.wordpress.org/reference/functions/register_block_type/#return)

[WP\_Block\_Type](https://developer.wordpress.org/reference/classes/wp_block_type/)|false The registered block type on success, or false on failure.

## [Use](https://developer.wordpress.org/reference/functions/register_block_type/#return):

You can pass custom $attributes which can be used both on editor and front-end in render\_callback :

register\_block\_type( 'my\_namespace/my\_block', [

'render\_callback' => 'render\_callback',

'attributes' => [

'some\_string' => [

'default' => 'default string',

'type' => 'string'

],

'some\_array' => [

'type' => 'array',

'items' => [

'type' => 'string',

],

]

]

] );

You can load block.json directly:

register\_block\_type( dirname(\_\_FILE\_\_) . '/block.json' );

register\_block\_type(

'my-custom-blocks/calendar',

**array**(

'attributes' => **array**(

'align' => **array**(

'type' => 'string',

'enum' => **array**( 'left', 'center', 'right', 'wide', 'full' ),

),

'day' => **array**(

'type' => 'integer',

),

'month' => **array**(

'type' => 'integer',

),

'year' => **array**(

'type' => 'integer',

),

),

'render\_callback' => 'render\_block\_my\_custom\_blocks\_calendar',

'editor\_script' => 'calendar-editor-js',

'editor\_style' => 'calendar-editor-css',

'script' => 'calendar-frontend-js',

'style' => 'calendar-frontend-css',

)

);

# registerBlockType() to Create Blocks in WordPress javascript

I learned that this function is at the heart of Block Development in WordPress since you must use it to create custom blocks.

The creation of a block in WordPress (as of writing) is done via JavaScript (not PHP) using a function called registerBlockType().

## How to Access registerBlockType()

[This function](https://github.com/WordPress/gutenberg/blob/3a3c3d5dad79eee00baf09f98e7cfd5416a4e698/blocks/api/registration.js#L51-L125) lives inside of the [wp.blocks library](https://github.com/WordPress/gutenberg/tree/master/blocks), which contains helpful functions for block creation, as well as the core WordPress blocks themselves.

WordPress makes this function accessible as a global variable in the window object as wp.blocks.registerBlockType().

// Get registerBlockType() from wp.blocks in the global scope

const { registerBlockType } = window.wp.blocks;

## registerBlockType Name and Settings Parameters

The registerBlockType() function takes two parameters: name and settings.

* **name** [string] –  “Block names must contain a namespace prefix, include only lowercase alphanumeric characters or dashes, and start with a letter. Example: my-plugin/my-custom-block” [Source](https://github.com/WordPress/gutenberg/blob/3a3c3d5dad79eee00baf09f98e7cfd5416a4e698/blocks/api/registration.js#L64-L69)
* **settings** [Object] – The settings parameter is an object with several predefined properties that must be assigned for your block to work.  We will look at these in more depth.

// Two parameters, name and settings

registerBlockType( 'example-plugin/example-custom-block', {} );

## registerBlockType Settings

The registerBlockType() settings object has seven properties that include both meta information about the block as well as methods controlling the UI and functionality for the block in the editor and on the frontend.

* **title** [string] – The title setting gives your block a searchable, human readable name. It should be escaped using wp.i18n.\_\_().
* **category** [string] – The category setting determines under which heading a user can find your block. Options include “common”, “formatting”, “layout”, “widgets” and “embed.”
* **icon** [Dashicon|Element] – The icon setting for registerBlockType determines what icon will represent your custom block. Can use a WP Dashicon or custom SVG.
* **keywords** [Array] – The keyword setting provides three additional keyword / phrases that will display your block when searched for. Limited to 3.
* **attributes** [Object] – The attribute setting identifies what dynamic content we are using in our blocks. Several attribute options exist depending on what types of data we are using. Attribute settings are optional if your block uses no dynamic data. This data is then made available via props.attributes.name. [Read more](https://wordpress.org/gutenberg/handbook/reference/attributes/).
* **edit** [function] – The edit setting determines what be displayed in the editor area for the block. Must return a valid React element using wp.element.createElement() or JSX. Also accepts dynamic data as a props parameter.
* **save** [function] – The save setting determines what be displayed when the block is converted to content for the frontend. Must return a valid React element using wp.element.createElement() or JSX. Also accepts dynamic data as a props parameter.

The first four of these [title, category, icon and keywords] are more for the user and include meta information about the block itself.  The last few settings [attributes, edit, save] are a bit more complex and control the actual functionality of a block.

**Enqueueing Block Scripts and Styles**

To add the JavaScript and CSS required by our blocks, we can use two new WordPress hooks provided by Gutenberg:

* enqueue\_block\_editor\_assets
* enqueue\_block\_assets

These are only available if the Gutenberg plugin is active, and they work in a similar way to standard WordPress hooks for enqueueing scripts. However, they are intended specifically for working with Gutenberg blocks.

### Category

There are six block categories currently available:

1. TEXT

2. MEDIA

3. DESIGN

4. WIDGETS

5. THEME

6. EMBEDS

## Example of registerBlockType in Action

// Get helper functions from global scope

const { registerBlockType } = window.wp.blocks;

const { \_\_ } = window.wp.i18n;

// Use registerBlockType to create a custom block

registerBlockType(

'block-namespace/block-name',

{

// configuration object

title: \_\_( 'Block Title' ),

category: 'common',

icon: 'wordpress-alt',

keywords: [

\_\_( 'Example' ),

\_\_( 'Project' ),

\_\_( 'Code Samples' )

],

// Attributes set for each piece of dynamic data used in your block

attributes: {

exampleContent: {

type: 'array',

source: 'children',

selector: 'div.my-content',

},

},

// Determines what is displayed in the editor

edit: props => {

const onChangeContent = value => {

props.setAttributes( { exampleContent: value } );

};

return (

<div className={props.className}>

<Editable

tagname="div"

multiline="p"

className="my-content"

placeholder={ \_\_( 'Add your content…' ) }

value={props.attributes.exampleContent}

onChange={onChangeContent}

/>

</div>

);

},

// Determines what is displayed on the frontend

save: props => {

return (

<div className={props.className}>

{props.attributes.exampleContent}

</div>

);

},

},

);

**Block Structure**

1. Main plugin file (wp-block.php)
2. Block editor css file (editor.css)
3. **Editor scripts.js file(editor.js)**
4. Block frontend css file(style.css)

As long as we import our main block file into **src/blocks.js** (as we did above) then we don't need to do any additional work. All JSX, ES6+, and Sass code will automatically be compiled into the following files:

* **dist/blocks.style.build.css**: styles for editor and front end
* **dist/blocks.build.js**: JavaScript for editor only
* **dist/blocks.editor.build.css**: styles for editor only

INSIDE: /local\_dev\_site/wp-content/plugins/my-block

npx create-guten-block my-block

├── plugin.php

├── package.json

└── **dist**

├── blocks.build.js

├── blocks.editor.build.css

└── blocks.style.build.css

**└── src**

├── **block**

├── block.js

├── editor.scss

└── style.scss

├── blocks.js

├── common.scss

└── init.php

registerBlockType(

    'my-unique-namespace/ultimate-block',

    {

        title: \_\_( 'The Best Block Ever', 'domain' ),

        icon: 'wordpress',

        category: 'common',

        keywords: [ \_\_( 'sample', 'domain' ), \_\_( 'Gutenberg', 'domain' ), \_\_( 'block', 'domain' ) ],

        edit: () => <h2>Welcome to the Gutenberg Editor!</h2>,

        save: () => <h2>How am I looking on the front end?</h2>

    }

);

**Simple Example :**

const { registerBlockType } = wp.blocks;

const { RichText } = wp.editor;

registerBlockType('myblock/rich-text-block', {

title: 'My Rich Text Block',

icon: 'welcome-write-blog',

category: 'common',

edit: function() {

....

},

save: function() {

....

}

});

**Step 1: Create a plugin to call up your block files**

=>wp-content/plugins/test-block.php

<?php

/\*\*

 \* Plugin Name: Test Plugin

 \* Author: John Doe

 \* Version: 1.0.0

 \*/

function loadMyBlock() {

  wp\_enqueue\_script(

    'my-new-block',

    plugin\_dir\_url(\_\_FILE\_\_) . 'test-block.js',

    array('wp-blocks','wp-editor'),

    true

  );

}

add\_action('enqueue\_block\_editor\_assets', 'loadMyBlock');

### Step 2: Register your block and configure its attributes

With the PHP file ready, it’s time to set up the test-block.js JavaScript file. Go ahead and create this file within the same plugin directory as plugins /test-block.php

/\* This section of the code registers a new block, sets an icon and a category, and indicates what type of fields it'll include. \*/

wp.blocks.registerBlockType('brad/border-box', {

  title: 'Simple Box',

  icon: 'smiley',

  category: 'common',

  attributes: {

    content: {type: 'string'},

    color: {type: 'string'}

  },

/\* This configures how the content and color fields will work, and sets up the necessary elements \*/

  edit: function(props) {

    function updateContent(event) {

      props.setAttributes({content: event.target.value})

    }

    function updateColor(value) {

      props.setAttributes({color: value.hex})

    }

    return React.createElement(

      "div",

      null,

      React.createElement(

        "h3",

        null,

        "Simple Box"

      ),

      React.createElement("input", { type: "text", value: props.attributes.content, onChange: updateContent }),

      React.createElement(wp.components.ColorPicker, { color: props.attributes.color, onChangeComplete: updateColor })

    );

  },

  save: function(props) {

    return wp.element.createElement(

      "h3",

      { style: { border: "3px solid " + props.attributes.color } },

      props.attributes.content

    );

  }

})

/////////////////////////////////////////////////////////////////////////////////

## Creating a block category

### Adding the category :

**add\_filter( 'block\_categories\_all' , function( $categories ) {**

**// Adding a new category.**

**$categories[] = array(**

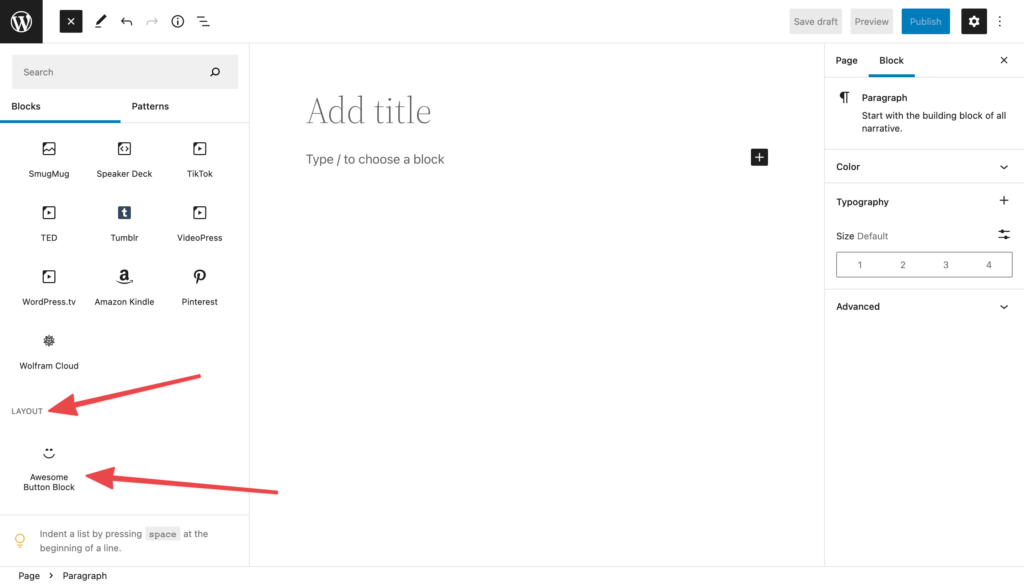
**'slug' => 'custom-layout-category',**

**'title' => 'Layout'**

**);**

**return $categories;**

**} );**

****

**WordPress Gutenberg Components Guid**

**Input Components**

Text input (TextControl)

Textarea (TextareaControl)

Rich Text (RichText)

Checkbox (CheckboxControl)

Radio Buttons (RadioControl)

Dropdown Select (SelectControl)

Toggle (ToggleControl)

Range (RangeControl)

Colorpicker (ColorPicker)

Date & Time Picker (DateTimePicker, DatePicker)

Font Size Picker (FontSizePicker)

Angle Picker (AnglePickerControl)

Resize (ResizableBox)

**Interactive Components**

Button (Button)

Button Group (ButtonGroup)

Icon (Icon, Dashicon)

Spinner (Spinner)

Tooltip (Tooltip)

Advanced Tooltip (Popover)

Toggleable Advanced Tooltip (Dropdown)

Dropdown Menu (MenuGroup, MenuItem)

Notice (Notice)

Modal (Modal)

Clickable External Link (ExternalLink)

Color Preview (ColorIndicator)

Content wrapper: Placeholder (Placeholder)

Content wrapper: Disabled (Disabled)

Content wrapper: Generic input (BaseControl)

Inspector Section (PanelBody)

Inspector Section Content Wrapper (PanelRow)