## Getting Started

The goal of this training is to help developers transition from the Classic Editor to the Block Editor. Before we can get into our block training, we need to set up a new WordPress site in Pantheon, load our training theme, and familiarize ourselves with the new files. Follow these steps to get the training up and running:

**Important Note:** **Our new scaffold requires WSL to run on PC’s. If you do not have WSL installed, please contact IT to get setup before continuing this training.**

1. Set up a sandbox WordPress site within Pantheon. This site does not need to be part of The Mx Group organization.
2. Once the site is set up, clone the [wordpress-training](https://docs.google.com/document/d/14KeqkELnMa1t2wkP2quEartAenhd1yb41z4T3x6rQ_s/edit#) repos from Bitbucket, drop it in the themes directory and activate it.
3. Now that the theme is in place it’s time to fire up our build tools. Open your WSL terminal and change directories until you are in the root of the theme. Run the following command **“npm run start”**.
4. If the build system runs successfully without errors, you’re ready to move onto step #5. If you got errors when you ran the command, that means WSL or one of the node dependencies are not set up correctly. Reach out to a teammate for assistance. If they are unable to assist, create a ticket with IT.
5. Now that the build command has run successfully, you just need to kickoff the watch command. In the WSL terminal, run **“npm run watch”**. Now as you save CSS, JS, and PHP files, it will trigger the build system to run and compile our code.
6. You’re now ready to start the training!

## Scaffold Overview

Okay! You have the training site setup. Now it’s time to take a look at our new scaffold and file structure. Although a large portion of the files have not changed from our past theme, there are several new files or directories to review. I have created a visual representation of the theme files below. The ones marked in white will be covered in this training. If you would like to learn more about the base files that come with a WordPress theme, head over to the Codex to check out their documentation on [WordPress Template Hierarchy](https://developer.wordpress.org/themes/basics/template-hierarchy/).

FED Scaffold Files

|

| -- assets

| -- CSS

| -- 1\_global

| -- 2\_components

| -- 3\_blocks

| -- example-block

| -- example-block-admin.css

| -- example-block.css

| -- \_admin.css

| -- \_index.css

| -- 4\_pages

| -- admin

| -- \_index.css

| -- base-blocks.css

| -- admin-styles.css

| -- block-admin-styles.css

| -- login-styles.css

| -- styles.css

| -- fonts

| -- images

| -- js

| -- 1\_global

| -- 2\_components

| -- 3\_blocks

| -- \_nested-blocks

| -- example-nested-block

| -- attributes.js

| -- edit.js

| -- index.js

| -- save.js

| -- example-block

| -- attributes.js

| -- edit.js

| -- index.js

| -- save.js

| -- admin

| -- block-editor.js

| -- blocks-blacklist.js

| -- admin-scripts.js

| -- blocks.js

| -- scripts.js

| -- svg

| -- videos

| -- config

| -- webpack.config.common.js

| -- webpack.config.dev.js

| -- webpack.config.prod.js

| -- webpack.csettings.js

| -- dist

| -- css

| -- js

| -- images

| -- includes

| -- classes

| -- post-types

| -- example-post-type.php

| -- blocks.php

| -- core.php

| -- overrides.php

| -- template-functions.php

| -- template-tags.php

| -- languages

| -- ThemeName.pot

| -- node\_modules

| -- template-parts

| -- components

| -- sidebar.php

| -- tests

| -- vendor

| -- .browserlistrc

| -- .editorconfig

| -- .eslintignore

| -- .eslintrc.json

| -- .gitignore

| -- .npmrc

| -- .nvmrc

| -- .stylelintrc

| -- 404.php

| -- archive.php

| -- babel.config.js

| -- category.php

| -- composer.json

| --composer.lock

| -- footer.php

| -- functions.php

| -- header.php

| -- index.php

| -- LICENSE.md

| -- package-lock.json

| -- package.json

| -- page-style-guide.php

| -- page.php

| -- phpcs.xml

| -- postcss.config.js

| -- README.md

| -- screenshot.png

| -- search.php

| -- style.css

| -- tag.php

| -- wpacceptance.json

### Configuration Files

For the most part, you will never have to touch the configuration files listed below. However, it is helpful to understand what these files are and what they do.

* **config** - This directory houses all the webpack config files that power our build system. You can disregard all the files in this directory besides webpack.settings.js.
  + **webpack.settings.js** - This file allows us to control the entry and exit points for our build system. If you need to enqueue a new css or js file, you can add the entry point in the “entries” section of this file.
* **.browserslistrc** - This config file tells our build tool which browsers we support and will drive autoprefixer settings.
* **.editorconfig** - This config file standardizes our code formatting across our team.
* **.eslintignore** -This config file tells our linting system which files to ignore.
* **.eslintrc.json** - This config is where we set our global variables and control our JS linting rules. We can use this file to limit what eslint rules are enforced and how strictly.
* **.npmrc** - This config file ensures that NPM is installed and required for this build system and sets the engine to true.
* **.nvmrc** - This config file declares what version of node to use in this project.
* **.stylelintrc** - This config is where we control our CSS linting rules. We can use this file to limit what stylelint rules are enforced and how strictly.
* **babel.config.js** - This config file declares which Babel presets to use. In our situation, we are simply extending 10Up’s Babel presets.
* **composer.json & composer.lock** - These composer files ensure that the necessary “vendor” packages are installed when the command “npm install” is run. The files handle PHP linting as well as automated testing. At this time we are not using PHP linting or the automated testing, but we may start using these features in the future.
* **package-lock.json & package.json** - These files ensure that all the necessary node packages are installed when the command “npm install” is run.
* **phpcs.xml** - This config file is similar to the babel.config.js file. We are simply extending 10Up’s PHP Code Sniffer settings. Again, PHPCS is used for PHP linting and automated testing.
* **postcss.config.js** - This file allows us to add PostCSS plugins to our build system. New PostCSS plugins can be added to the “plugins” section at the top of this file.
* **wpacceptance.json** - This config file is used for automated testing. Again, we are not using this feature at this time so you should not have to edit this file.

### Miscellaneous Directories & Files

Similar to the configuration files, there is another group of directories that you should familiarize yourself with, but you will not have to interact with these files.

* **languages** -This folder houses all the language translations for the site.At this time, we have not built a multilingual site using WordPress’ default translation functionality so you can ignore this folder currently.
* **tests** - This folder houses files that help with automated testing as well as accessibility testing. Because we are not using this functionality at this time, you can ignore these files.
* **vendor** - Similar to the “node\_modules” folder, this folder houses all the dependencies that come from the composer.json file.
* **node\_modules** - You will not see this directory until you run “npm Install”. Once this command is run, this folder will appear with all the necessary node packages.
* **dist** - Once “npm install” is run, the dist folder should appear in the root of our theme. This is where our compiled CSS & JS files are output. These files should already be enqueued via the “core.php” and “blocks.php” files in the “includes” directory.

### Includes Directory & Functions.php

In the next section, we will take a closer look at the “includes” directory and files. We will also see how this section relates to functions.php in the root of the theme.

* **includes** - This folder houses key php files that help set up our WordPress site. From this directory, we can create PHP classes, register custom post types, setup and enqueue stylesheets/scripts, as well as create global PHP functions or template tags.
  + **classes** - Classes are used to develop object-oriented PHP. Although we do not use this folder very often, this is where we would place more complex WordPress functionality.
  + **post-types** - This folder houses any custom post types that need to be registered on the site. Each post type should be split out into its own file and any related taxonomies should be registered in that file. Our theme includes an example-post-type.php file which can be used as a base.
  + **blocks.php** - This file handles the enqueue of all block related styles and scripts. Because block styles and scripts need to be added to both the admin and the frontend, these enqueues are separated from our sites main enqueues which are contained in the next file, “core.js”
  + **core.php** - This is the main file that handles our WordPress site setup. This file declares style and script enqueues as well as registers menus.
  + **overrides.php** - This file removes and disables unnecessary WordPress functionality like emoji support, etc. This reduces the amount of elements enqueued or loaded and helps improve performance.
  + **template-functions.php** - This is where all WordPress filters and actions are housed. Like template-tags.php, our theme comes with several filters and actions. Please take some time to look these over so you can use them if needed.
  + **template-tags.php** - This is where all global PHP functions should be housed. These functions should be accessible via WordPress partials, etc. Our theme comes with several template tags by default. Take some time to familiarize yourself with these functions so that you can take advantage of them.

Now that you understand what files make up the “includes” directory, it is time to understand how they relate to functions.php. All WordPress sites have a functions.php file. Some themes load all their classes, post-types, template functions, and template tags in a single functions.php file. Because this file can get really large, we chunk out the code to make it scalable and easier to maintain. With that being said, you will see that our functions.php file defines our global constants and then calls the PHP housed in the “includes” directory.

### CSS Files

Because our new theme is built around the block editor, we have to separate our frontend styles from our admin styles. With that being said, we have several new CSS files in the base of the directory “admin-styles.css”, “block-admin-styles.css”, “login-styles.css”, and “styles.css”. There are also two new folders under CSS, “3\_blocks” and “admin”.

* **3\_blocks** - This folder houses all block styles.
  + **example-block** - Each block should have its own directory nested within “3\_blocks”
    - **example-block-admin.css** - Admin styles for the block go here.
    - **example-block.css** - Frontend styles go here.
  + **\_admin.css** - Block admin styles should be imported here.
  + **\_index.css** - Block frontend styles should be imported here.
* **admin** - This is the general entry point for admin styles unrelated to a specific block. This is where all global admin styles should be located.
  + **\_index.css** - admin styles should be imported here.
  + **base-block.css** - This base file is similar to the base file used on the frontend. It includes all global admin styles.
* **admin-styles.css** - This file imports all admin CSS partials that are not related to an individual block (“admin/\_index”).
* **block-admin-styles.css** - This file imports all admin CSS related to individual blocks (“3\_blocks/\_admin”).
* **login-styles.css** - This file contains all styling related to the login page.
* **styles.css** - this file contains all frontend CSS imports including frontend block styles.

### JS Files

In this section, we will take a closer look at our JS files. Again, we need to split out our frontend scripts from our admin scripts. With that being said, the file structure in this directory has changed slightly.

* **3\_blocks** - This folder houses all normal blocks as well as nested blocks.
  + **\_nested-blocks** - All nested blocks should be kept separate from regular blocks. You will learn more about nested blocks later in the training.
  + **example-block** - This directory will include all the necessary files to make up a block (attributes.js, edit.js, index.js, save.js).
* **admin** - This folder contains all admin related scripts. By default, our theme starts with 2 admin files block-editor.js and blocks-blacklist.js.
  + **block-editor.js** - This file runs some basic js related to our link component.
  + **blocks-blacklist.js** - This file unregisters all core blocks.
* **admin-scripts.js** - This file imports all admin JS partials related to the admin folder.
* **blocks.js** - This file imports all block related components, blocks, and nested blocks.
* **scripts.js** - this file imports all remaining frontend scripts.

### Dist Files

As mentioned above, all CSS and JS code is compiled and output to the “dist” directory. This is setup by default in our build tools and the files should already be enqueued correctly for you. In addition to the CSS/JS folders, you will see an “images” directory. Our scaffold also includes built-in image optimization. If the build tool is in watch mode when you add an image to the “assets/images/” folder, it should automatically optimize the image and output the updated version to the “dist/images” folder. Please make sure your relative paths in your PHP and CSS reference the file in the dist directory.

* **dist** - This folder houses all compiled CSS and JS as well as optimized images.
  + **css** - All compiled CSS files are output here.
  + **js** - All compiled JS files are output here.
  + **images** - Optimized images are output here.

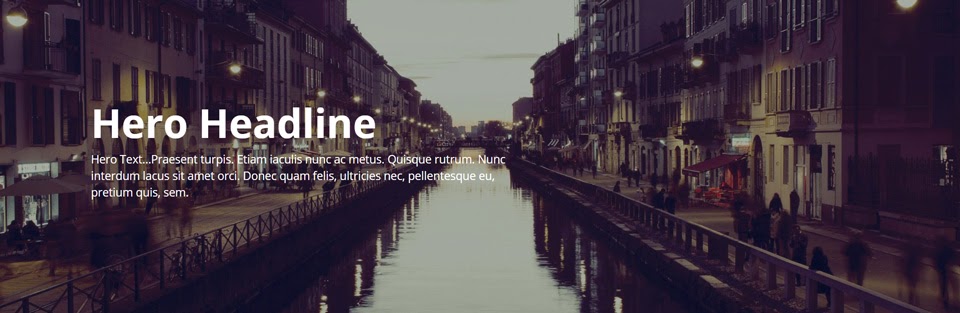
Okay! We’ve finally reviewed all the new files in our theme. Now that we have a better understanding of how these files work, It’s time to jump into our Block Editor training!

## Building Our First Block

The goal of this section is to take you through the process of building a WordPress Block. We’ll start by examining the anatomy of a basic block. From there, we’ll dissect the key parts and explore how each piece functions. This training will also cover basic concepts of ES6, React, and JSX. Once you have a strong understanding of basic blocks, we’ll see how Nested Blocks can take our projects to the next level.

**Important Note: The finalized code for the blocks in this training can be found in a folder called “training-solutions” which is located in the root of the theme. Because copy and pasting from the code blocks in this document can be hit or miss, it may be easier to work out of these files.**

In this training we’ll be building a basic hero block. This block will be made up of a headline, paragraph text, and an image. Here is the design we will be building:



Here is a link to the background image used in this training:

<https://drive.google.com/file/d/1qqVTCjCZxATDL65hZrpgmWEvopzIMZdK/view?usp=sharing>

### Block Registration

Custom blocks are just an extension of WordPress’ built-in block functionality. When creating a custom block, we’re extending a WordPress class to create and register a new block. As you will see below, we start by creating a registerBlockType function. This function may look a little different than you’re used to because the Block Editor uses the common.js syntax. This syntax allows us to define functions and make them exportable to other files.

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

});

When registering a block, we provide the namespace and the slug for the block. In the example above, ‘wordpress-training’ is the namespace of our project and ‘hero-block’ is the slug.

Every custom block consists of the 8 key elements which are listed below:

1. **title** - The title of the block which is displayed in the admin.
2. **description** - This description is also visible in the admin section and provides more context about the block
3. **icon** - This icon will be displayed in several places of the admin. An icon can be loaded in two ways:
   1. [WordPress’ Dashicons](https://developer.wordpress.org/resource/dashicons/)
   2. Import a custom svg icon
4. **category** - Blocks can be categorized and block access can be limited based on the category.
5. **keywords** - We can provide keywords that help the user find blocks via the block search field in the admin
6. **attributes** - We’ll definitely go over this section in more detail later, but for now, think of attributes as the same thing as “state” in React. We define attributes based on the elements needed to make up a block. Each attribute is a placeholder to capture and store block data
7. **edit** - Think of the “Edit” function as the area where all admin functionality exists. This is where we build the admin layout for our block and include all necessary components to capture our data. Once a block is edited and saved, the data is stored in attributes and passed to the frontend via the “Save” function.
8. **save** - The “Save” function handles the frontend layout of the block. This is where we create our markup and pull in the attribute data. Again, we will go over the “Edit” and “Save” functions in greater detail later in the training.

**Important Note:** For the purposes of this training, all code will be placed in a single file. However in real-world scenarios, we should split our code for attributes, icons, and “Edit/Save” functions into their own files. These files will be imported into the index.js file.

Block Metadata

Elements 1 - 5 make up the majority of the block’s metadata. Let’s add these elements to our block below.

/\*\*

\* WordPress dependencies

\*/

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

const { registerBlockType } = wp.blocks;

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

title: \_\_('Hero Block', 'wordpress-training'),

description: \_\_('Hero Block Description', 'wordpress-training'),

icon: 'megaphone',

category: 'custom-blocks',

keywords: [

\_\_('Hero', 'wordpress-training'),

\_\_('Block', 'wordpress-training'),

\_\_('Hero Block, 'wordpress-training')

],

});

**Important Note:** In the section above the block registration, we have an import statement. This code pulls in functionality from other WordPress dependencies. In this situation, we’re importing the “\_\_” element from the i18n component so we can [internationalize](https://developer.wordpress.org/block-editor/developers/internationalization/) our text. You’ll see that we’re using “\_\_” as a prefix to the text elements above. In this example we’re also loading the megaphone icon from [WordPress Dashicons](https://developer.wordpress.org/resource/dashicons/).

### Attributes

As mentioned above, attributes are used to store and pass block data. All data captured during the “Edit” function must be stored in our block state if we want to access it on the frontend. In the next step we define each data point as an attribute. As you will see below, attributes take the shape of an object.

/\*\*

\* WordPress dependencies

\*/

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

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

title: \_\_('Hero Block', 'wordpress-training'),

description: \_\_('Hero Block Description', 'wordpress-training'),

icon: 'format-image',

category: 'custom-blocks',

keywords: [

\_\_('Hero', 'wordpress-training'),

\_\_('Block', 'wordpress-training'),

\_\_('Hero Block, 'wordpress-training')

],

attributes: {

headline: {

type: 'string',

default: '',

selector: '.hero-block\_\_headline',

},

imgAlt: {

type: 'string',

default: '',

},

imgID: {

type: 'number',

default: '',

},

imgURL: {

type: 'string',

default: '',

},

text: {

type: 'string',

source: 'html',

selector: '.hero-block\_\_text',

},

},

});

Based on our design, we’ll need headline text, paragraph text, and an image. The most important rule when defining attributes is declaring a “type”. Here is a list of valid [attribute types](https://developer.wordpress.org/block-editor/developers/block-api/block-attributes/). From here, we have the option to set “default” data. You will also see that some of the attributes above have a “selector” value. This value is not required for all attributes, however, if you are displaying text, you will need to define a “selector” to bind that attribute data to. This will make more sense once we get to the “Save” function.

### New Coding Concepts

Before we can dig into the “Edit” function, we need to go over several new coding concepts used in the Block Editor. We will start with ES6, then move onto React concepts. Lastly, we will go over JSX, what it is, and how it helps streamline the code in our blocks and components.

#### ES6 Concepts

##### Arrow Functions

The first concept we will cover is [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions). Arrow functions are a new JavaScript syntax that reduces the amount of code we have to write. Because arrow functions can be written a variety of ways, it isn’t always easy to understand the code unless we understand the difference between implicit and explicit returns. In the first example below, you will see the difference between traditional functions and arrow functions.

/\* Traditional Function - Explicit Return \*/

function (a) {

return a + 100;

}

/\* Arrow Function - Explicit Return \*/

(a) => {

return a + 100;

}

/\* Arrow Function - Implicit Return \*/

(a) => a + 100;

/\* Arrow Function - Implicit Return, Multiple Parameters \*/

(a, b) => a + b + 100;

/\* Arrow Function - Implicit Return, Single Parameter \*/

a => a + 100;

As you will see above, the difference is purely syntax related. In the first two functions we are “explicitly” defining the return value by using the term “return”. The three functions below “implicitly” define the return function by dropping the braces and removing the term “return”. In the last example, you will see that the parenthesis have been removed from the arguments. We can only remove the parentheses if there is a single argument. If there is more than one, we have to include the parentheses.

##### Destructuring

The next concept we’ll cover is [object and array destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment). Destructuring makes it easier to extract data from objects/arrays in one statement. You’ve already seen this concept in action when we imported the internationalization functionality at the top of our file. We used destructuring to pull the “\_\_” element out of an object called wp. Let’s take a closer look:

/\* Visual representation of the wp object \*/

wp: {

i18n: {  
 \_\_: {

Function to internationalize text

}

}

}

/\* Example of destructuring using normal JavaScript \*/

const wp = wp.object; // Assign a variable to the main object

const i18n = wp.i18n; // Dig into main object and assign i18n to a variable

const \_\_ = i18n.\_\_; // Dig deeper and assign \_\_ to a variable

/\* Example of ES6 destructuring \*/

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

As you can see destructuring reduces the amount of code we have to write. Because the Block Editor passes data using a React object called “props”, this concept will be used a lot. Next, we will take a closer look at another ES6 concept called Template Literals and how they are used when building blocks.

##### Template Literals

Template Literals makes it easier to concatenate variables and strings in JavaScript. Be sure to check out the documentation on Template Literals [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals). In the meantime, here is a brief example of Template Literals in action:

/\* Example of concatenating in normal JavaScript \*/

const text = 'variable text';

console.log( 'This is a string of text and this is ' + text );

/\* Example using ES6 template literals \*/

console.log( `This is a string of text and this is ${text}` );

#### In the example above, we wrap all the text in backticks ``. This tells the browser to interpret the text as a string. Unlike concatenation, template literals do not require you to switch back and forth between single and double quotes. Also, you no longer have to use the ‘+’ symbol to tie strings and variables together. Instead we can use this new syntax, ${text}, to call variables within strings.

#### When we get to the basic concepts of JSX, we’ll see some real-world examples of template literals. For now, let’s take a closer look at a few key React concepts before we start applying them to our hero block.

#### React Concepts

It’s important to understand that the Block Editor is built on React, but WordPress uses its own syntax so there are differences between the React documentation and the Block Editor Handbook. The key difference between the two is that React requires a render() function and a return() function to display components. In WordPress we only need to use the return() function.

**Important Note:** WordPress components and the “Edit/Save” functions can only return **one** element. If you need to return multiple elements/components, they must be wrapped in a single div or a [React Fragment](https://reactjs.org/docs/fragments.html).

##### Props

As mentioned above, we’ll be using React “Props” to pass data. This data can be passed via the “Edit” and “Save” function as well as via components. By default, the “Props” object includes several data points. Be sure to **console.log( props )** inside the “Edit” function to see everything available. In the meantime, we will go over the ones that are used most:

* **attributes** - Once you define all your attributes, they’re automatically stored inside props as an object called “attributes”.
* **className** - When a block is registered, WordPress automatically creates a prop called “className”. This class name is a combination of the namespace and block slug we covered earlier. In our hero block, the class name generated is “wp-block-wordpress-training-hero-block”. We will apply different classes to our block for styling purposes.
* **setAttributes** - This prop is actually a utility function and it will play a key role in setting our attribute data or saving our block state. All components will use the setAttributes function to save associated data.

In the example below, we create a function called “Edit”. Then we begin by destructuring our props.

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

/\* Block metadata goes here... \*/

edit: ( props ) => {

/\* Checkout all default props that come with a block \*/

console.log( props );

/\* Destructure props object \*/

const { attributes, className, setAttributes } = props;

/\* Destructure attributes object \*/

const { headline, imgAlt, imgID, imgURL, text } = attributes;

},

} );

##### Now that we have access to all the variables and data we need, the next step is to structure our return function and start adding components to our editor interface.

##### Components

The great part about the Block Editor is that we have an entire [component library](https://developer.wordpress.org/block-editor/components/) at our disposal. We can mix and match components to build our interface. In the example below, we import the TextControl component from the library, we’ve destructured all our props, and added a basic return statement.

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components; // Import from component library

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

/\* Block metadata goes here \*/

edit: ( props ) => {

const { attributes, className, setAttributes } = props;

const { headline, imgAlt, imgID, imgURL, text } = attributes;

return (

<div className={`${className} hero-block editor-hero-block`}>

<TextControl

className='hero-block\_\_headline'

value={headline}

onChange={(headline) => setAttributes({ headline })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

</div>

);

},

});

##### Inside the return statement, we added some additional markup to give the layout structure. We also add the code for the [TextControl](https://developer.wordpress.org/block-editor/components/text-control/) component. Let’s take a closer look at the component itself so we can get a better understanding of how it works.

/\*\*

\* TextControl Component

\*/

<TextControl // CamelCased Component Name

className='hero-block\_\_headline' // Prop

value={headline} // Prop

onChange={(headline) => setAttributes({ headline })} // Prop

placeholder={\_\_('Placeholder Text', 'wordpress-training')} // Prop

keepPlaceholderOnFocus='true' // Prop

/>

##### 

##### All React components take the shape of a normal HTML element. However, the main difference is the component name should be capitalized and camel-cased. Each of the line items in this component look like HTML attributes, but they’re technically React “Props”. When using Props in a component, we define how the component renders and behaves in the editor area.

##### Let’s take a closer look at the props being set in this TextControl:

* **className** - this prop allows us to set our own classes on editor elements for styling purposes.
* **value** - we will always set the “value” of the component to its corresponding attribute/state. In this case, we set the value to our “headline” text. Now whenever the state of the “headline” text changes, this value will update in the component.
* **onChange** - The most important prop of any component is onChange. As you can see, the onChange prop is actually passing the “setAttributes” arrow function. This built-in arrow function implicity sets the state for this component.
* **placeholder** - defines placeholder text for the field in the admin area.
* **keepPlaceholderOnFocus** - This prop tells the component how to behave on focus.

##### Okay, we’ve started building our “Edit” function, we imported our first component, we destructured our variables, and we added our first component. At this point, our code is taking shape, but it looks strange compared to the normal JavaScript we’re used to writing. That means it’s time to go over JSX and how it’s different.

#### JSX Concepts

JSX stands for “JavaScript XML” and it’s a syntax extension that allows us to write HTML inside our JavaScript files. All JSX code must be executed within a return() statement. Any code written outside the return statement will be executed like plain JavaScript. What’s great about JSX, is our ability to switch back-and-forth between HTML and JavaScript very easily.

##### Basic Concepts

Similar to PHP, JSX uses a special syntax to indicate when the browser should interpret code as JavaScript or HTML. There are two main ways to call JavaScript within JSX:

1. **JSX Brace’s** - Like the <?php ?> tag tells the browser to execute PHP code, braces {} indicate which code should be interpreted as JavaScript. There are multiple examples below using this method to call the “headline” variable.
2. **Template Literals** - We can also use template literals to tell the browser when to execute JS. In the example below, we enter JS with an opening brace {, then we call “className” using a template literal variable of ${className}, lastly we add a few more classes using strings and then we exit JS with the closing brace }.

/\*\*

\* PHP Syntax

\* Example of executing PHP within an HTML element

\*/

<h1 class="hero-block\_\_headline"><?php echo $headline; ?></h1>

/\*\*

\* JSX Syntax

\* Example of executing JavaScript within a JSX element

\*/

return ( // Returns single JSX div element

/\* No code allowed here \*/

/\* JSX Brace’s and Template Literal Example (directly below) \*/

<div className={`${className} hero-block editor-hero-block`}>

<TextControl

className='hero-block\_\_headline'

value={headline} // JSX Brace’s Example

onChange={(headline) => setAttributes({ headline })} // JSX Brace’s

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

</div>

/\* No code allowed here \*/

);

##### Helpful Snippets

Now that we can switch back and forth between JSX and HTML, below are some helpful code snippets that will be used extensively within React and JSX. To learn more about conditional rendering, check out the [documentation](https://reactjs.org/docs/conditional-rendering.html).

/\*\*

\* React Syntax for Conditional Logic

\*/

/\* If Statement - If headline text exists, display h1 element \*/

{ headline.length > 0 &&

<h1 class="hero-block\_\_headline">{headline}</h1>

}

/\* If/Else Long-form Ternary Statement \*/

{ '2' === buttons ? (

<div className='button-wrap'>

<a className='btn btn--primary' href={ buttonURL }>

{ buttonText }

</a>

<a className='btn btn--primary' href={ buttonURL }>

{ buttonText }

</a>

</div>

) : (

<div className='button-wrap'>

<a className='btn btn--primary' href={ buttonURL }>

{ buttonText }

</a>

</div>

) }

/\* If/Else Inline Ternary Statement (className) \*/

<TextControl

className={`hero-block\_\_headline ${modifier ? 'headline--bold' : ''}`}

value={headline}

onChange={(headline) => setAttributes({ headline })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

### The Edit Function

Now that we have a better understanding of the new coding concepts required, it’s time to put everything together and complete our “Edit” function. In the example below, you’ll see we added two more components to handle the paragraph text and the hero image. We also added some wrapper elements to flesh out the structure a little more.

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

const { RichText } = wp.blockEditor;

/\*\*

\* Custom Component Imports

\*/

import ImagePicker from '../../2\_components/image-picker'; // Import custom component

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

/\* Block metadata goes here \*/

edit: ( props ) => {

const { attributes, className, setAttributes } = props;

const { headline, imgAlt, imgID, imgURL, text } = attributes;

return (

<div className={`${className} hero-block editor-hero-block`}>

<div className='hero-block\_\_copy'>

<TextControl

className='hero-block\_\_headline'

tagName='h1'

value={headline}

onChange={(headline) => setAttributes({ headline })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

<RichText

className='hero-block\_\_text'

tagName='div'

multiline='p'

value={text}

onChange={(text) => setAttributes({ text })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

</div>

<div className='hero-block\_\_image'>

<div className='image-picker'>

<label htmlFor='ImagePicker'>

Background Image:

<ImagePicker

imgURL={imgURL}

imgID={imgID}

imgAlt={imgAlt}

setAttributes={setAttributes}

/>

</label>

</div>

</div>

</div>

);

},

});

Above, we’ve added another component to the mix. This component is called [RichText](https://developer.wordpress.org/block-editor/developers/richtext/) and it takes the place of the WYSIWYG field used in the classic editor. We’ll be using this component a lot so let’s take a closer look below.

/\*\*

\* Attributes

\*/

attributes: {

text: { // RichText Field

type: 'string',

source: 'html', // Allows component to render HTML elements like <p> tags

selector: '.hero-block\_\_text',

},

},

/\*\*

\* RichText Component

\*/

<RichText

className='hero-block\_\_text'

tagName='div'

multiline='p' // Tell’s editor to wrap returns in new <p> tags

value={text}

onChange={(text) => setAttributes({ text })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

Although the RichText and TextControl components are very similar, there are a few slight differences. The first occurs in the way we define our attributes. As you can see above, RichText requires an attribute property called “source”. We set this to HTML so that the RichText component can render nested HTML elements like <strong> tags or <p> tags. The second difference is the addition of the “multiline” prop. This property tells the component how to handle “returns” or “line breaks”. In this situation we are telling the editor to wrap every return in a <p> tag...which is how the WYSIWYG editor currently works. The last difference you will notice for the RichText component takes place in the “Save” function, so we’ll cover that a little bit later in the training.

The last component added to our “Edit” function handles the selection of the background image. Unlike the previous examples which use built-in WordPress components, this is a **custom component** and it uses several components from the library to accomplish its functionality. You will see all the component imports at the top of the file. When using the MediaUpload component, we have to define the valid media types. These media types take the form of an array and are also set at the top of this file.

/\*\*

\* WordPress dependencies

\*/

const { Button, Icon, Tooltip } = wp.components;

const { MediaUpload, MediaUploadCheck } = wp.blockEditor;

const { Fragment } = wp.element;

const ALLOWED\_MEDIA\_TYPES = ['image'];

/\*\*

\* Image Picker Component

\*

\* @param {Array} props passed in props

\* @return {void} background image component

\*/

const ImagePicker = (props) => {

const { imgAlt, imgID, imgURL, setAttributes } = props;

/\*\*

\*

\* onSelectImage

\*/

const onSelectImage = (img) => {

setAttributes({

imgID: img.id,

imgURL: img.url,

imgAlt: img.alt,

});

};

/\*\*

\*

\* onRemoveImage

\*

\*/

const onRemoveImage = () => {

setAttributes({

imgID: null,

imgURL: null,

imgAlt: null,

});

};

return (

<Fragment>

{!imgID ? (

<MediaUploadCheck>

<div className='components-panel\_\_item media-wrap'>

<MediaUpload

allowedTypes={ALLOWED\_MEDIA\_TYPES}

onSelect={onSelectImage}

value={imgID}

id='bg-image'

render={({ open }) => (

<Tooltip text='Add Image'>

<Button className='add-media' onClick={open}>

<Icon icon='admin-media' />

Add Image

</Button>

</Tooltip>

)}

/>

</div>

</MediaUploadCheck>

) : (

<div className='components-panel\_\_item image-wrapper'>

<img src={imgURL} alt={imgAlt} />

<Tooltip text='Remove Image'>

<Button className='remove-media' onClick={onRemoveImage}>

<Icon icon='no-alt' />

</Button>

</Tooltip>

</div>

)}

</Fragment>

);

};

export default ImagePicker;

We won’t go over this component line-by-line, but the goal is to create a simple interface for the user to select images from the media library. Because the user needs the ability to add and remove images, you’ll see we have two event handler functions above. The first sets the image attributes and the second sets them back to null. One function fires when the user clicks on the media upload button and the other when the user clicks the remove image button. At the bottom of the file is an export statement. Again, this is common.js syntax and it allows us to export the component and import it into other JS files. As you become more familiar with the component library and React, this example will make more sense.

Okay now for a brief recap before we move onto the “Save” function. We have registered our block and added our metadata (including attributes). We have built out our Block Editor interface via the “Edit” function and now, if we make changes to the fields in the admin, we should see the values update in realtime. The next step is to set up our “Save” function and build out the frontend markup.

### The Save Function

As you will see below, the “Save” function is much less complex than the “Edit” function. Like the ”Edit” function, we use destructuring to pull our data out of the props object. Below the destructuring, you will see we set our background image using [React Inline Styling](https://reactjs.org/docs/dom-elements.html#style). This allows us to define CSS properties in a JS object and apply them to the DOM element via the “style” attribute. Please keep in mind that this technique should be used sparingly and only for dynamic CSS properties. All other block styles should be kept in our CSS files.

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

const { RichText } = wp.blockEditor;

/\*\*

\* Custom Component Imports

\*/

import ImagePicker from '../../2\_components/image-picker'; // Import custom component

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

/\* Block metadata goes here \*/

/\* Edit function goes here \*/

save: ( props ) => {

const { headline, text, imgURL} = props.attributes;

const bgImageStyles = {

backgroundImage: `url( ${imgURL} )`,

};

return (

<section className='hero-block'>

<div className='hero-block\_\_background' style={bgImageStyles}>

<div className='hero-block\_\_inner'>

{ headline.length > 0 &&

<h1 className='hero-block\_\_headline'>{headline}</h1>

}

{ text.length > 0 && (

<RichText.Content

className='hero-block\_\_text'

tagName='div'

value={text}

/>

) }

</div>

</div>

</section>

);

},

});

The rest of the “Save” function consists of a return statement and JSX. You will see we’re using conditional logic to determine if the headline and text elements should be displayed. Inside each of those if statements, we are calling the values that have been stored in our block state.

Now that we are calling our block data on the frontend, this is probably a good time to circle back to our Attributes and close the loop on the “selector” properties. Earlier, we mentioned you need to define a selector when working with text data. This selector tells the Block Editor which HTML element should display the text. You will see in our attributes section, the selectors are “.hero-block\_\_headline” and “hero-block\_\_text” respectively. These values match up with the classNames used in the “Save” function. It’s important that these values match up so the Block Editor knows where to display the text.

/\*\*

\* Attributes

\*/

attributes: {

headline: {

type: 'string',

default: '',

selector: '.hero-block\_\_headline',

},

text: {

type: 'string',

source: 'html',

selector: '.hero-block\_\_text',

},

},

/\*\*

\* Save Markup Example

\*/

{ headline.length > 0 &&

<h1 className='hero-block\_\_headline'>{headline}</h1> // className matches selector

}

{ text.length > 0 && (

<RichText.Content

className='hero-block\_\_text' // className matches selector above in Attributes

tagName="div"

value={text}

/>

) }

The last area of the “Save” function that we need to go over in more detail is the RichText Component. Earlier we mentioned that this component works a little different in the “Save” function. As you can see above, we don’t call the “text” attribute data inside an HTML element like we did with the Headline. Instead, we call a frontend component called <RichText.Content /> which pulls in all the appropriate markup defined in the editor. We declare a tagName to tell the editor what to wrap our content in. In this case all our RichText content will be wrapped in a <div> tag and keyboard returns will be wrapped in <p> tags.

Congratulations! You have completed your first block! At this point, your block should be up and running and the final code should look like this:

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

const { RichText } = wp.blockEditor;

/\*\*

\* Custom Component Imports

\*/

import ImagePicker from '../../2\_components/image-picker'; // Import custom component

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/hero-block', {

title: \_\_('Hero Block', 'wordpress-training'),

description: \_\_('Hero Block Description', 'wordpress-training'),

icon: 'format-image',

category: 'custom-blocks',

keywords: [

\_\_('Hero', 'wordpress-training'),

\_\_('Block', 'wordpress-training'),

\_\_('Hero Block, 'wordpress-training')

],

attributes: {

headline: {

type: 'string',

default: '',

selector: '.hero-block\_\_headline',

},

text: {

type: 'string',

source: 'html',

selector: '.hero-block\_\_text',

},

imgAlt: {

type: 'string',

default: '',

},

imgID: {

type: 'number',

default: '',

},

imgURL: {

type: 'string',

default: '',

},

},

edit: ( props ) => {

const { attributes, className, setAttributes } = props;

const { headline, text, imgAlt, imgID, imgURL } = attributes;

return (

<div className={`${className} hero-block editor-hero-block`}>

<div className='hero-block\_\_copy'>

<TextControl

className='hero-block\_\_headline'

tagName='h1'

value={headline}

onChange={(headline) => setAttributes({ headline })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

<RichText

className='hero-block\_\_text'

tagName='div'

multiline='p'

value={text}

onChange={(text) => setAttributes({ text })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus="true"

/>

</div>

<div className='hero-block\_\_image'>

<div className='image-picker'>

<label htmlFor='ImagePicker'>

Background Image:

<ImagePicker

imgURL={imgURL}

imgID={imgID}

imgAlt={imgAlt}

setAttributes={setAttributes}

/>

</label>

</div>

</div>

</div>

);

},

save: ( props ) => {

const { headline, text, imgURL} = props.attributes;

const bgImageStyles = {

backgroundImage: `url( ${imgURL} )`,

};

return (

<section className='hero-block' style={bgImageStyles}>

<div className='hero-block\_\_inner'>

{ headline.length > 0 &&

<h1 className='hero-block\_\_headline'>{headline}</h1>

}

{ text.length > 0 && (

<RichText.Content

className='hero-block\_\_text'

tagName='div'

value={text}

/>

) }

</div>

</section>

);

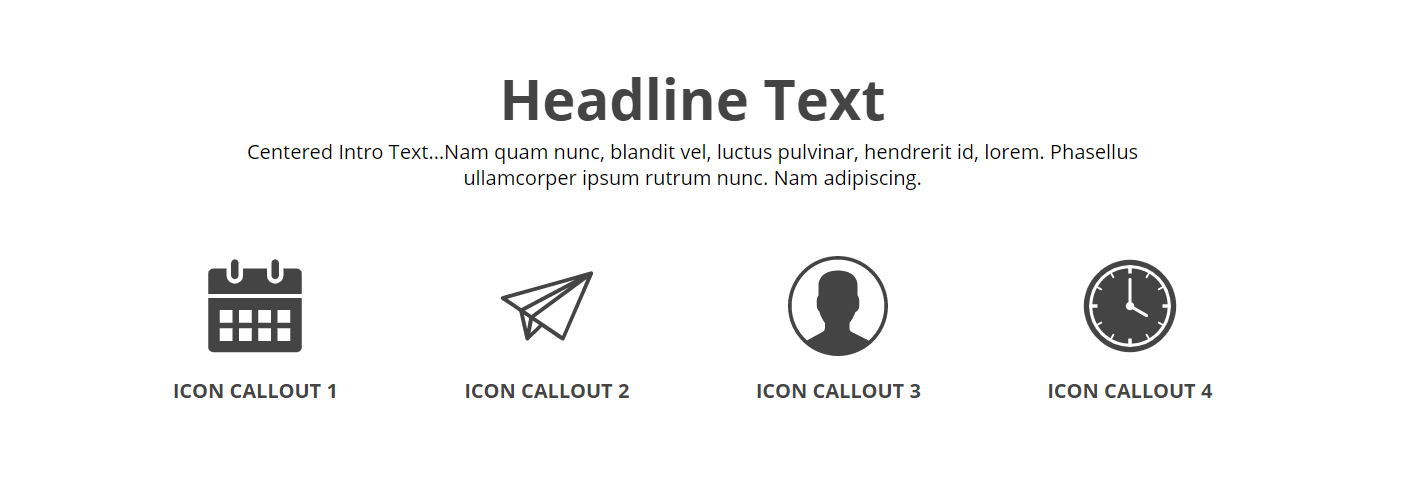
},

});

## 

## Building a Nested Block

In this section of the training we’ll build a block with repeatable content. Repeatable content is a vital part of any WordPress site. Currently, we use repeater fields in ACF to display repeatable content like cards, sliders, callouts, etc. In the Block Editor, we can create repeatable blocks and nest them within existing parent blocks to accomplish similar functionality. For this exercise we’ll be creating the Icon Callouts block shown below:

****

For the parent block, we will need headline and text fields. For the nested block, we will need an icon and callout text. Let’s start by setting up the nested block first and then we’ll integrate it with the parent block.

Here is a link to the icon assets for this section:

<https://drive.google.com/drive/folders/1Kt9z76jB193I9QZdHTQM0WLfOAUwLhT2?usp=sharing>

### Nested Block

Nested blocks are similar to basic blocks with one exception. Nested blocks have an additional meta field called “parent”. This field is used to define which parent blocks have access to the nested block. You’ll see below that we set the parent value to “[‘wordpress-training/icon-callouts’]”. This value matches the info used to register the parent block. As you can see this value is an array and you can define multiple parent blocks.

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

/\*\*

\* Custom Component Imports

\*/

import ImagePicker from '../../2\_components/image-picker'; // Import custom component

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/icon-callout', {

title: \_\_('Icon Callout', 'wordpress-training'),

description: \_\_('Icon Callout Description', 'wordpress-training'),

icon: 'admin-comments',

category: 'custom-blocks',

keywords: [

\_\_('Icon', 'wordpress-training'),

\_\_('Callout', 'wordpress-training'),

\_\_('Icon Callout, 'wordpress-training')

],

parent: ['wordpress-training/icon-callouts'], // Define Parent Block(s)

attributes: {

imgAlt: {

type: 'string',

default: '',

},

imgID: {

type: 'number',

default: '',

},

imgURL: {

type: 'string',

default: '',

},

text: {

type: 'string',

source: 'html',

selector: '.icon-callout\_\_text',

},

},

edit: ( props ) => {

const { attributes, className, setAttributes } = props;

const { imgAlt, imgID, imgURL, text } = attributes;

return (

<div className={`${className} icon-callout editor-icon-callout`}>

<div className='icon-callout\_\_icon'>

<div className='image-picker'>

<label htmlFor='ImagePicker'>

Background Image:

<ImagePicker

imgURL={imgURL}

imgID={imgID}

imgAlt={imgAlt}

setAttributes={setAttributes}

/>

</label>

</div>

</div>

<div className='icon-callout\_\_copy'>

<TextControl

className='icon-callout\_\_text'

tagName='h4'

value={text}

onChange={(text) => setAttributes({ text})}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

</div>

</div>

);

},

save: ( props ) => {

const { imgALT, imgURL, text } = props.attributes;

return (

<div className='icon-callout'>

{ imgURL.length > 0 && (

<div className='icon-callout\_\_icon'>

<img src={imgURL)} alt={imgALT} />

</div>

) }

{ text.length > 0 &&

<h4 className='icon-callout\_\_text'>{text}</h4>

}

</div>

);

},

});

We won’t go over this card in detail because it’s similar to the block we built in the first training exercise. Take note of the “parent” value as it’s key to creating nested blocks.

### 

### Parent Block

Now that we have the nested block in place, it’s time to set up the parent block and pull everything together. Let’s start by adding our imports and block metadata. You’ll see below that we’re importing a new component called “InnerBlocks” from the wp.blockEditor object. The term “Inner Blocks” and “Nested Blocks” are basically interchangeable and you can check out the documentation [here](https://developer.wordpress.org/block-editor/tutorials/block-tutorial/nested-blocks-inner-blocks/). This import allows us to use the nested blocks within our parent block.

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

const { RichText } = wp.blockEditor;

const { InnerBlocks } = wp.blockEditor; // Import InnerBlocks Component

const ALLOWED\_BLOCKS = ['wordpress-training/icon-callout']; // Define allowed Nested Blocks

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/icon-callouts', {

title: \_\_('Icon Callouts', 'wordpress-training'),

description: \_\_('Icon Callouts Description', 'wordpress-training'),

icon: 'format-chat',

category: 'custom-blocks',

keywords: [

\_\_('Icon', 'wordpress-training'),

\_\_('Callouts', 'wordpress-training'),

\_\_('Icon Callouts, 'wordpress-training')

],

attributes: {

headline: {

type: 'string',

default: '',

selector: '.hero-block\_\_headline',

},

text: {

type: 'string',

source: 'html',

selector: '.icon-callouts\_\_text',

},

},

/\* Edit function goes here \*/

/\* Save function goes here \*/

});

Another key difference between a nested block and a parent block is that we need to define which nested blocks are allowed. In the example above, you’ll see we’re defining an array called “ALLOWED\_BLOCKS”. This is where we can list all the nested blocks accessible to the parent block. In the following example, we flesh out our “Edit” function and you can see how “ALLOWED\_BLOCKS” relate to the InnerBlocks component.

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

const { RichText } = wp.blockEditor;

const { InnerBlocks } = wp.blockEditor; // Import InnerBlocks Component

const ALLOWED\_BLOCKS = ['wordpress-training/icon-callout']; // Allowed nested blocks

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/icon-callouts', {

/\* Block metadata goes here \*/

edit: ( props ) => {

const { attributes, className, setAttributes } = props;

const { headline, text } = attributes;

return (

<section className={`${className} icon-callouts editor-icon-callouts`}>

<div className='icon-callouts\_\_copy'>

<TextControl

className='icon-callouts\_\_headline'

tagName='h2'

value={headline}

onChange={(headline) => setAttributes({ headline })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

<RichText

className='icon-callouts\_\_text'

tagName='div'

multiline='p'

value={text}

onChange={(text) => setAttributes({ text })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus="true"

/>

</div>

<div className='icon-callouts\_\_wrap'>

<InnerBlocks allowedBlocks={ALLOWED\_BLOCKS} />

</div>

</section>

);

},

});

Well that was pretty easy! We created a block and attributes to handle the main headline and text. Then we just called the InnerBlocks component and passed a prop called “allowedBlocks”. This will pull our nested blocks into the admin and WordPress takes care of the rest of the functionality.

To complete the parent block, all we have left is to build our save function. Like the edit function, this is easy! First we create the normal markup for the headline and text. Then we create a wrapper element for styling purposes. The last step is to call the frontend version of the InnerBlocks component, <InnerBlocks.Content />. This frontend component is similar to the RichText component we have been using in previous blocks.

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

const { RichText } = wp.blockEditor;

const { InnerBlocks } = wp.blockEditor;

const ALLOWED\_BLOCKS = ['wordpress-training/icon-callout'];

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/icon-callouts', {

/\* Block metadata goes here \*/

/\* Edit function goes here \*/

save: ( props ) => {

const { attributes, className} = props;

const { headline, text } = attributes;

return (

<div className='icon-callouts'>

<div className='icon-callouts\_\_inner'>

{ headline.length > 0 &&

<h2 className='icon-callouts\_\_headline'>{headline}<h2>

}

{ text.length > 0 &&

<div className='icon-callouts\_\_text'>{text}</div>

}

<div className='icon-callouts\_\_wrap'>

<InnerBlocks.Content />

</div>

</div>

</div>

);

},

});

Now that we have completed the “Save” function. Let’s put it all together so you have the final code for reference:

/\*\*

\* WordPress dependencies

\*/

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

const { TextControl } = wp.components;

const { RichText } = wp.blockEditor;

const { InnerBlocks } = wp.blockEditor; // Import InnerBlocks Component

const ALLOWED\_BLOCKS = ['wordpress-training/icon-callout']; // Define allowed Nested Blocks

/\*\*

\* Register block

\*/

export default registerBlockType('wordpress-training/icon-callouts', {

title: \_\_('Icon Callouts', 'wordpress-training'),

description: \_\_('Icon Callouts Description', 'wordpress-training'),

icon: 'format-chat',

category: 'custom-blocks',

keywords: [

\_\_('Icon', 'wordpress-training'),

\_\_('Callouts', 'wordpress-training'),

\_\_('Icon Callouts, 'wordpress-training')

],

attributes: {

headline: {

type: 'string',

default: '',

selector: '.hero-block\_\_headline',

},

text: {

type: 'string',

source: 'html',

selector: '.icon-callouts\_\_text',

},

},

edit: ( props ) => {

const { attributes, className, setAttributes } = props;

const { headline, text } = attributes;

return (

<div className={`${className} icon-callouts editor-icon-callouts`}>

<div className='icon-callouts\_\_copy'>

<TextControl

className='icon-callouts\_\_headline'

tagName='h2'

value={headline}

onChange={(headline) => setAttributes({ headline })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus='true'

/>

<RichText

className='icon-callouts\_\_text'

tagName='div'

multiline='p'

value={text}

onChange={(text) => setAttributes({ text })}

placeholder={\_\_('Placeholder Text', 'wordpress-training')}

keepPlaceholderOnFocus="true"

/>

</div>

<div className='icon-callouts\_\_wrap'>

<InnerBlocks allowedBlocks={ALLOWED\_BLOCKS} />

</div>

</div>

);

},

save: ( props ) => {

const { attributes, className} = props;

const { headline, text } = attributes;

return (

<div className={`${className} icon-callouts`}>

<div className='icon-callouts\_\_inner'>

{ headline.length > 0 &&

<h2 className='icon-callouts\_\_headline'>{headline}<h2>

}

{ text.length > 0 &&

<div className='icon-callouts\_\_text'>{text}</div>

}

<div className='icon-callouts\_\_wrap'>

<InnerBlocks.Content />

</div>

</div>

</div>

);

},

});

## Conclusion

Thank you for taking the time to complete this training. We know that this document does not cover every aspect of the Block Editor, but our goal is to get our developers up-to-speed with the basic concepts as quickly as possible.

If you come across grammar or spelling errors, please feel free to update this document. If you felt any section of this training was unclear, irrelevant, or redundant please let us know. We want these documents to evolve and stay relevant. If you believe there is an important concept missing from the training, please let us know so we can figure out the best way to communicate that info to the team.