WordPress Gutenberg

Components Guide

a white pixel

About This Guide

When WordPress version 5 shipped with the new Javascript-based block editor, Gutenberg, the code has been under rapid development. As a result of that there has been a lack of documentation on this topic.

As a developer who works with WordPress daily, both as a job and as a hobby, I had a hard time embracing the new block editor without a lot of trying and failing. Along the way notes, a-ha moments, and code snippets were jotted down. As an attempt to compile an overview of useful and re-usable components available in Gutenberg's packages, this guide wrote itself.

This guide focuses on the most common and re-usable components available in the WordPress Gutenberg library, which includes most types of form elements and user interaction elements. Examples are text inputs, checkboxes, buttons, colorpickers, dropdown menus, modals, notices, and useful content wrappers. Using the components that are available in WordPress anyway reduce the need to code and style your own components, and it also ensures coherent design.

You'll find code examples for all components mentioned in this guide. The examples assumes some knowledge and experience in creating your own custom blocks – such as fetching attribute values and updating them.

The overview of component's props is not a comprehensive guide in absolutely all available props. I have included the most important, commonly used, and most useful props. Wherever there are more props than listed, it's mentioned in the text. You can then click the Github link (Github icon next to the component's name) to go to the Gutenberg repository. In most cases you'll arrive at a readme file where you might find documentation on the props.

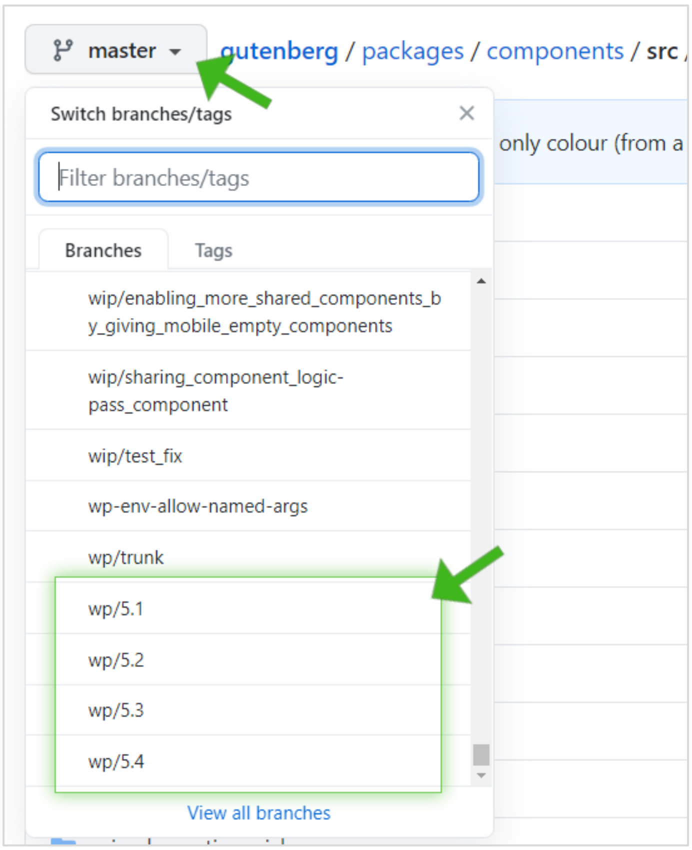
All code in this guide is written in Javascript ES6 / ES2015+. Keep in mind that you need Babel or similar to transform the script into ES5.

This guide is written and tested for **WordPress 5.4.2** (summer of year 2020).

Github / Source Code

WordPress keeps Gutenberg's source code in a publicly accessible Github repository: https://github.com/WordPress/gutenberg

For all components in this guide you will find a clickable Github icon next to its name. The links go directly to the component's source code in this reposiroty, and in most cases you'll arrive at the component's readme file that usually contains some documentation. However what I've found is that the readme files are not always kept up to date, nor do they always include all props available.



master

gutenberg / packages / components / src /

Switch branches/tags

Filter branches/tags

Branches | Tags

wip/enabling_more_shared_components_by_giving_mobile_empty_components

wip/sharing_component_logic-pass_component

wip/test_fix

wp-env-allow-named-args

wp/trunk

wp/5.1

wp/5.2

wp/5.3

wp/5.4

View all branches

only colour (from a

Keep in mind the repository's branches. All links in this guide refers to the `master` branch. But the code in the `master` branch is not reflecting precisely the version in the WordPress version you are currently running.

The repository have branches for major WordPress version releases; `wp/5.1`, `wp/5.2`, `wp/5.3`, and `wp/5.4`. Switch to these branches to see the code available in those WordPress versions.

WordPress theme or plugin developers would find most use of the code inside the folder `'packages'`. All the components are destructured from the packages inside this folder. For example you'll find the package folders `'components'`, `'block-editor'`, and `'element'`. Most of the components covered in this guide is inside the `components` package.

If you are interested in looking at WordPress Gutenberg's default blocks (Paragraph, Heading, Cover, and so on) you will find them inside the repository folder `packages/block-library/src`.

Table of Contents

Input Components

In this chapter you'll find the most common inputs. Inputs are GUI elements that expect the user to input or select some kind of value.

Most of input components resides in the package `wp.components`.

Commonly shared props

There are some shared traits with these components; they all expect props for the current value and an event for updating the value. These props are usually named `value` and `onChange`, respectively, unless otherwise stated.

Keep in mind that for most inputs to work, ie. change its value, the `onChange` (or otherwise named) prop must be provided with the proper value update action. For custom blocks this usually means updating the value of an attribute.

Another common shared prop is `label`. As the name states it will display a helpful text connected to the input, usually above. Most, but not all components support this so in those cases you need to manually create a label element if necessary.

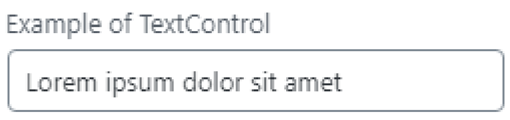
Finally another common shared prop is `className`. Most of the components support adding a custom class name to the input or its wrapper. This helps you to target the component for styling.

Text input

TextControl

wp.components

TextControl is a component for a standard text input (`<input type="text" ../>`) which lets users enter and edit text in a single-line GUI.



Props

TextControl accepts the common shared props value, onChange, label, and className. The props value and onChange are necessary for the input to be editable. The component supports additional props (see readme documentation), but some worth mentioning are;

type	(String) (optional) Set the type attribute of the input. Examples are "number", "email" and "url". Defaults to "text".

Code

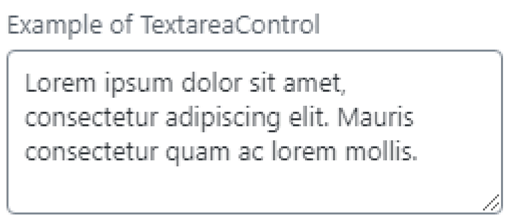
```
// Destructure component
const { TextControl } = wp.components;

// Basic usage
<TextControl
      label={__('Example of TextControl', 'awhitepixel')}
      value={props.attributes.stringAttribute}
      onChange={(val) => props.setAttributes({stringAttribute: val})}
/>
```

Textarea

TextareaControl

wp.components

TextareaControl is a component for rendering a <textarea> element that allows users to enter and edit text over multiple lines. A textarea has no text formatting options (if you are looking for this, check the RichText component instead. But keep in mind that RichText is not supported inside Inspector).



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris consectetur quam ac lorem mollis.

Props

TextareaControl accepts the common shared props value, onChange, and label. The props value and onChange are necessary for the input to be editable. The component supports additional props (see readme documentation), but some worth mentioning are;

rows	(String) (optional) Define the number of rows the textarea should contain. Default is "4".

Code

```
// Destructure component
const { TextareaControl } = wp.components;

// Basic usage
<TextareaControl
	label={__('Example of TextareaControl', 'awhitepixel')}
	value={props.attributes.stringAttribute}
	onChange={(val) => props.setAttributes({stringAttribute: val})}
/>
```

Rich Text

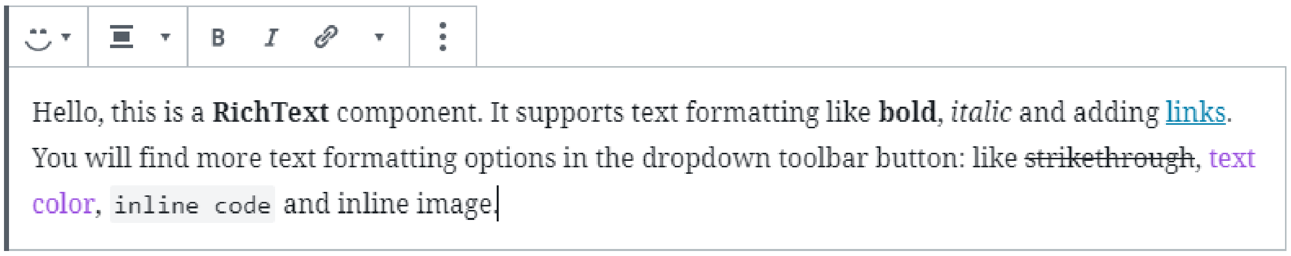
`RichText`

`wp.blockEditor`

The `RichText` component renders an UI to edit content with support for text formatting and URLs.

Keep in mind that the `RichText` component is meant to be used within the block editor. It will not work properly inside the Inspector.

Inside the block's save function you need to use `<RichText.Content>` to correctly save and render the content.



Hello, this is a **RichText** component. It supports text formatting like **bold**, *italic* and adding links. You will find more text formatting options in the dropdown toolbar button: like ~~strikethrough~~, text color, `inline code` and inline image.

Props

`RichText` accepts the common shared props `value` and `onChange`. The props `value` and `onChange` are necessary for the input to be editable. The component supports additional props (see readme documentation), but some worth mentioning are;

`tagName`	(String) (optional) The wrapping tag of the editable element. Inline tag elements (e.g. `'span'`) are not supported. Examples: `'h1'`, `'h2'`, `'div'`, or 'p'. Default is `'div'`.
`placeholder`	(String) (optional) If set a placeholder when the field is empty will be displayed. If you want the placeholder to stay even if the field is focused/selected, set the prop `keepPlaceholderOnFocus` to true.
`multiline`	(Boolean) (optional) By default a line break tag (`<br/>`) is added for each newline. If you want to support paragraphs when pressing Enter set this to true.

`preserveWhiteSpace`	(Boolean) (optional) By default tabs, newline and space characters are collapsed into a single space. To avoid this set this prop to true as to keep any whitespace as is.
`allowedFormats`	(Array of strings) (optional) A prop that allows you to control which text formatting options are available. By default all registered formats are allowed. Default formats are named as following: Bold: '`core/bold`', Italic: '`core/italic`', URL: '`core/link`', Strikethrough: '`core/strikethrough`', Inline code: '`core/code`', Inline image: '`core/image`', and Text color: '`core/text-color`'.

Code

```
// Destructure component
const { RichText } = wp.blockEditor;

// Basic usage
<RichText
       value={props.attributes.stringAttribute}
       onChange={(val) => props.setAttributes({stringAttribute: val})}
/>

/* If you are using RichText for adding a title, set tagName prop to the desired title
tag. It also makes sense to disable certain text formatting options that is not fit for a
title. In the example below RichText is used to make a h2 title with only italic,
strikethrough and text color available: */
<RichText
       value={props.attributes.stringAttribute}
       onChange={(val) => props.setAttributes({stringAttribute: val})}
       tagName="h2"
       allowedFormats={['core/italic', 'core/strikethrough', 'core/text-color']}
/>
```

Checkbox

`CheckboxControl`

`wp.components`

A component for rendering a checkbox `<input type="checkbox">` element that allows users to select one or more items from a set.

If you want a more fancy UI that looks like a toggle, see "Toggle" input.

Example of CheckboxControl

Props

`CheckboxControl` accepts the common shared props `onChange` and `label`. Note that the `value` prop for this component is `checked`, and the value needs to be a boolean. The props `checked` and `onChange` are necessary for the input to be editable. Also note that the `label` prop will appear inline after the checkbox, instead of a block-styled label above the input.

The component supports additional props (see readme documentation), but some worth mentioning are;

`heading`	(String) (optional) An additional prop for displaying a label before the checkbox, as the prop `label` will be displayed inline after the checkbox. Keep in mind that standard WordPress styling will not make the `heading` element block styled. As default it will appear inline *before* the checkbox. Some styling fixes or manual label may be necessary.

Code

```
// Destructure component
const { CheckboxControl } = wp.components;

// Basic usage
<CheckboxControl
	label={__('Example of CheckboxControl', 'awhitepixel')}
	checked={props.attributes.booleanAttribute}
	onChange={(val) => props.setAttributes({booleanAttribute: val})}
/>
```

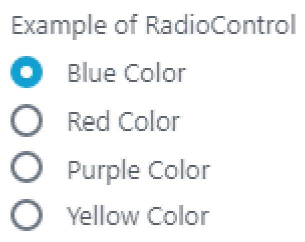
Radio Buttons

`RadioControl`

`wp.components`

A component for rendering a set of options in the form of radio buttons. You provide a set of choices in which the user can choose only one.

The props and handling of this component is pretty much identical to `SelectControl`. The difference is only in the GUI representation.



Props

`RadioControl` accepts the common shared props `onChange` and `label`. Note that the value prop for this component is `selected`. The props `selected` and `onChange` are necessary for the input to be editable.

The component supports additional props (see readme documentation), but some worth mentioning are;

`options`	(Array of objects) (optional) An array of choices for radio buttons. Each choice should be an object with the following properties; `label`: the visible text after the radio button, and `value`: the internal value of the choice and what is passed to the `onChange` event.

Code

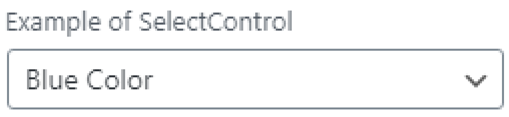
```
// Destructure component
const { RadioControl } = wp.components;

// Basic usage
<RadioControl
       label={__('Example of RadioControl', 'awhitepixel')}
       selected={props.attributes.stringAttribute}
       onChange={(val) => props.setAttributes({stringAttribute: val})}
       options={[
              { label: __('Blue Color', 'awhitepixel'), value: 'blue' },
              { label: __('Red Color', 'awhitepixel'), value: 'red' },
              { label: __('Purple Color', 'awhitepixel'), value: 'purple' },
              { label: __('Yellow Color', 'awhitepixel'), value: 'yellow' },
       ]}
/>
```

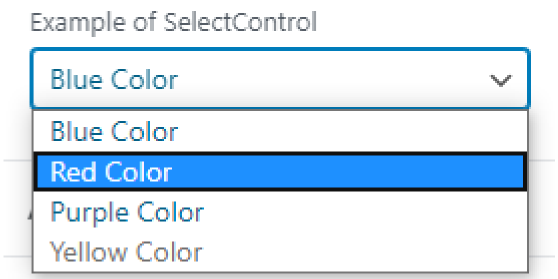
Dropdown Select

`SelectControl`

`wp.components`

A wrapper component for a `<select>` element that allows users to choose an option from a dropdown menu. A select is a good way to cleanly display several options for the user without displaying all of the available options at once.





Props

`SelectControl` accepts the common shared props `value`, `onChange`, and `label`. The props `value` and `onChange` are necessary for the input to be editable. The component supports additional props (see readme documentation), but some worth mentioning are;

`options`	(Array of objects) (optional) An array of choices for the dropdown. Each choice should be an object with the following properties; `label`: the visible text in the dropdown, and `value`: the internal value of the choice and what is passed to the `onChange` event. Optionally you can provide the property `disabled`: adds the "disabled" attribute to the option making the choice un-choosable.
`multiple`	(Boolean) (optional) If set to true the GUI changes into a multiselect (larger box) that allows users to hold CTRL or Shift to select multiple options. Keep in mind that this requires the value to be an *array of values* instead of a singular value.

Code

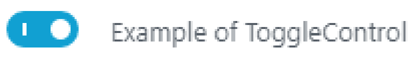
```
// Destructure component
const { SelectControl } = wp.components;

// Basic usage
<SelectControl
	label={__('Example of SelectControl', 'awhitepixel')}
	value={props.attributes.stringAttribute}
	onChange={(val) => props.setAttributes({stringAttribute: val})}
	options={[
		{ label: __('Blue Color', 'awhitepixel'), value: 'blue' },
		{ label: __('Red Color', 'awhitepixel'), value: 'red' },
		{ label: __('Purple Color', 'awhitepixel'), value: 'purple' },
		{ label: __('Yellow Color', 'awhitepixel'), value: 'yellow', disabled: true },
	]}
/>

// Example of multiselect:
// props.attributes.arrayAttribute = ['blue', 'purple'];
<SelectControl
	label={__('Example of multi-select', 'awhitepixel')}
	value={props.attributes.arrayAttribute}
	onChange={(val) => props.setAttributes({arrayAttribute: val})}
	options={[
		{ label: __('Blue Color', 'awhitepixel'), value: 'blue' },
		{ label: __('Red Color', 'awhitepixel'), value: 'red' },
		{ label: __('Purple Color', 'awhitepixel'), value: 'purple' },
		{ label: __('Yellow Color', 'awhitepixel'), value: 'yellow' },
	]}
	multiple={true}
/>
```

Toggle

`ToggleControl`

`wp.components`

A toggle is an user interface that allows the user to turn an option on or off.



Props

`ToggleControl` accepts the common shared props `onChange` and `label`. Note that the value prop for this component is `checked`, and that it must be a boolean. The props `checked` and `onChange` are necessary for the input to be editable.

Code

```
// Destructure component
const { ToggleControl } = wp.components;

// Basic usage
<ToggleControl
	label={__('Example of ToggleControl', 'awhitepixel')}
	checked={props.attributes.booleanAttribute}
	onChange={(val) => props.setAttributes({booleanAttribute: val})}
/>
```

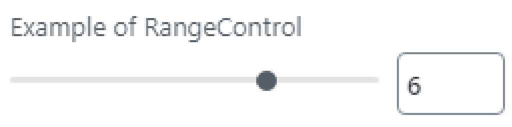
Range

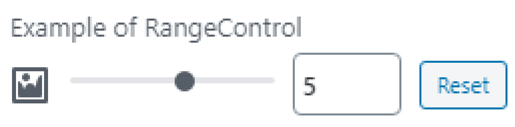
`RangeControl`

`wp.components`

`RangeControl` is a component for rendering a slider interface allowing the user to make a selection between a range of incremental values. The range must be between numbers.

Images below show two examples of `RangeControl`. The UI difference is controlled via props.





Props

`RangeControl` accepts the common shared props `onChange`, `value`, and `label`. The props `value` and `onChange` are necessary for the input to be editable.

The component supports additional props (see readme documentation), but some worth mentioning are;

`min` and `max`	(Number) (optional) Props for setting the minimum (`min`) and maximum (`max`) numbers allowed in the range.
`step`	(Number) (optional) Defines the stepping interval of the slider. Default is 1.
`beforeIcon` `afterIcon` `icon`	(String) (optional) Provide an icon to be displayed around the slider. `beforeIcon` appears before the slider, and `afterIcon` appears after. The `icon` prop is displayed above, next to the container title.
`allowReset`	(Boolean) (optional) If this prop is `true` a button to reset the slider is rendered. This is by default `false`. Use this in conjunction with the next prop for defining the reset value.

`resetFallbackValue`	(Number) (optional) Used in accordance with the `allowReset` prop above and define the value to revert to when the reset button is clicked.
`withInputField`	(Boolean) (optional) Set to `false` to not render a number input next to the slider. Default is `true`. NB: Was introduced in Gutenberg 7.5 and not part of WordPress 5.4.

Code

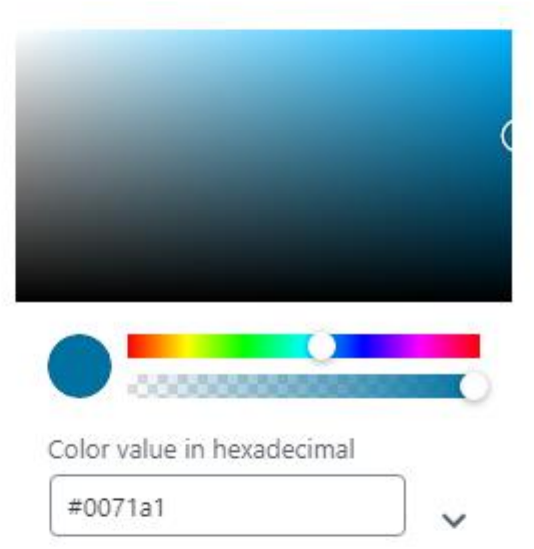
```
// Destructure component
const { RangeControl } = wp.components;

// Basic usage
<RangeControl
	label={__('Example of RangeControl', 'awhitepixel')}
	value={props.attributes.numberAttribute}
	onChange={(val) => props.setAttributes({numberAttribute: val})}
	min={1}
	max={8}
/>

// Example of adding additional GUI elements; an icon before the slider, a reset button,
and deactivate rendering the input field:
<RangeControl
	label={__('Example of RangeControl', 'awhitepixel')}
	value={props.attributes.numberAttribute}
	onChange={(val) => props.setAttributes({numberAttribute: val})}
	min={1}
	max={8}
	beforeIcon="format-image"
	allowReset={true}
	resetFallbackValue={6}
	withInputField={false}
/>
```

Colorpicker

`ColorPicker`

`wp.components`

A component for rendering a colorpicker, allowing the user to pick a color from the color wheel, or entering a hex code.



Props

The value prop for this component is `color` and the update event is `onChangeComplete`. The props `color` and `onChangeComplete` are necessary for the input to be editable.

Note that the value returned in the `onChangeComplete` event is an object with all properties to the chosen color (e.g. HSL, RGB values and more). Normally you would be interested in the `hex` property which contains the standard 6-length color hex code prefixed with a #.

`disableAlpha`	(Boolean) (optional) Define whether or not to render an opacity slider. Default `false`. Set to `true` to render the opacity slider.

Code

The example below saves the hex color code to a block attribute by referencing the `hex` property of the returned value in the `onChangeComplete` event. We can pass the hex value as value in the `color` prop - the `ColorPicker` component will understand what kind of value it is and display the correct color.

```
// Destructure component
const { ColorPicker } = wp.components;

// Basic usage
<ColorPicker
       label={__('Example of ColorPicker', 'awhitepixel')}
       color={props.attributes.stringAttribute}
       onChangeComplete={(val) => props.setAttributes({stringAttribute: val.hex})}
/>
```

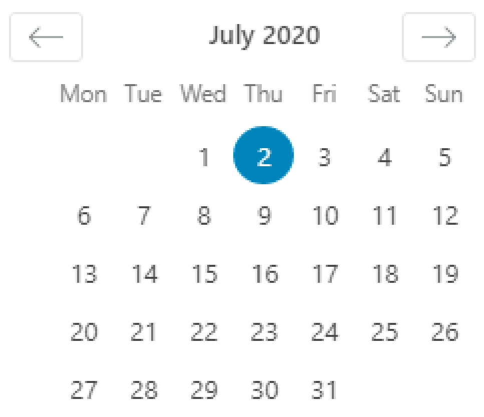
Date & Time Picker

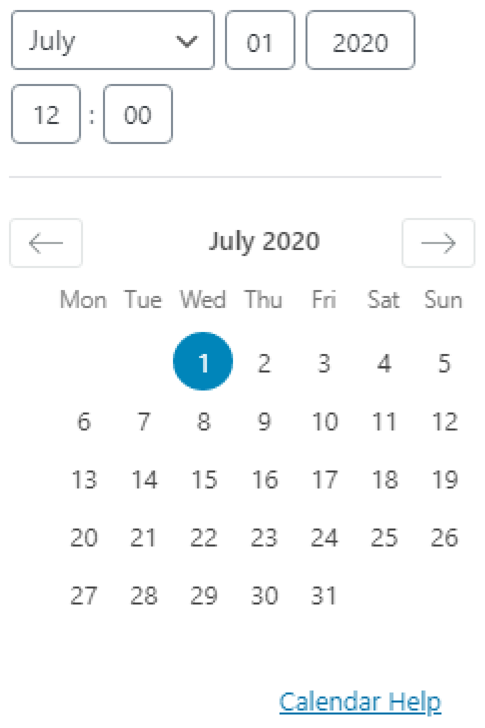
`DateTimePicker, DatePicker`

`wp.components`

There are three components available for selecting date and time. The `Datepicker` component renders a calendar where the user can pick a date. And the `DateTimePicker` component renders the same as `Datepicker` but allows the user to select a time at the day as well. WordPress' publish functionality is using the `DateTimePicker` component.

The third component, `Timepicker`, renders the timepicker. But I've excluded it as it's not possible to use without it rendering a datepicker as well.





Props

The props are the same for `DateTimePicker` and `DatePicker`, since `DateTimePicker` is simply rendering the `DatePicker` component in itself and passing all the props.

Both components accept the common shared props `onChange`. Note that the value prop for this component is `currentDate`. The current value and returned value from the update event is a string in the standard Javascript date format; "`YYYY-MM-DDTHH:mm:ss`". Pass `null` as value to `currentDate` to avoid setting an initial selected date.

There is no label prop so you need to manually render a label if necessary.

`is12Hour`	(Boolean) (optional) Boolean whether or not the time should be in 12 hour format or 24 hour format. If this prop is not provided, it will follow WordPress' date and time format settings. If this is `true` or WordPress' date and time settings are set to 12 hours, the component will render additional buttons for setting AM and PM.

Code

```
// Destructure the components, or just the one you want to use
const { DateTimePicker, DatePicker } = wp.components;

// Basic usage of DateTimePicker
<DateTimePicker
	currentDate={props.attributes.stringAttribute}
	onChange={(val) => props.setAttributes({stringAttribute: val})}
	is12Hour={false}
/>
```

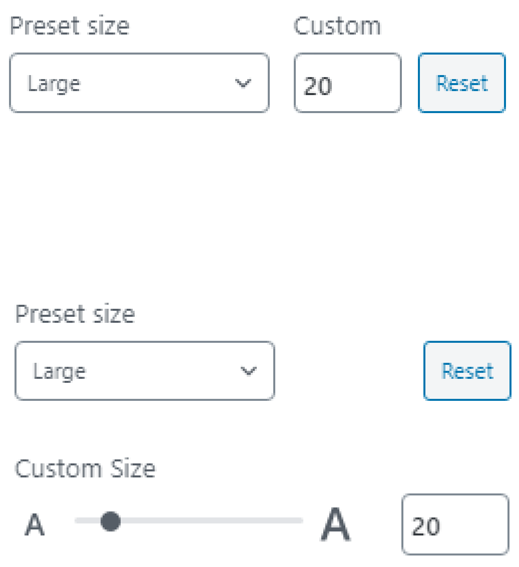
Font Size Picker

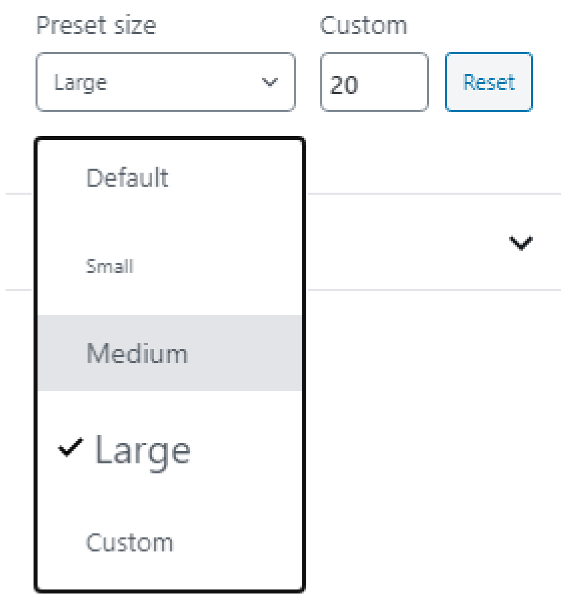
`FontSizePicker`

`wp.components`

`FontSizePicker` renders an UI for selecting a font size from a dropdown or a slider of predefined sizes, and optionally a number input to enter a custom size. You can define your own sizes and names, and the choices in the dropdown will render a preview of the font size.

Below are images of `FontSizePicker`. The elements are controllable via props.





Props

`FontSizePicker` accepts the common shared props `onChange` and `value`. Note that there are no label prop available so you need to manually render a label if necessary.

The component supports additional props (see readme documentation), but some worth mentioning are;

`fontSizes`	(Array of objects) (optional) An array of choices for font sizes. Each choice should be an object with the following properties; `size`: a number with the font size in px, `name`: the visible label (e.g.

	"Small"), and `slug`: an unique slug identifier which is used for class generation purposes. The value of `size`, a number, is what is passed in the `onChange` event and used to set initial value. Also note that the slugs `default` and `custom` are reserved and cannot be used.
`disableCustomFontSizes`	(Boolean) (optional) If `true` the user cannot set a custom font size. They are forced to pick one of the predefined sizes defined in the prop `fontSizes`.
`withSlider`	(Boolean) (optional) If set to `true` the component will render a slider below the fontsize picker. The input field is moved away from the top row and placed to the right of the slider.

Code

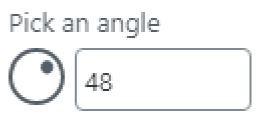
```
// Destructure component
const { FontSizePicker } = wp.components;

// Basic usage
<FontSizePicker
	value={props.attributes.numberAttribute}
	onChange={(val) => props.setAttributes({numberAttribute: val})}
/>

// The above will render only an input for entering a size and a reset button. If you want a dropdown to choose sizes from you need to provide the prop fontSizes. Below is an example of defining custom font sizes and adding the slider element

<FontSizePicker
	value={props.attributes.numberAttribute}
	onChange={(val) => props.setAttributes({numberAttribute: val})}
	fontSizes={[
		{ name: __('Small', 'awhitepixel'), slug: 'small', size: 10 },
		{ name: __('Medium', 'awhitepixel'), slug: 'medium', size: 14 },
		{ name: __('Large', 'awhitepixel'), slug: 'large', size: 20 },
	]}
	withSlider={true}
/>
```

Angle Picker

`AnglePickerControl`

`wp.components`

A component for rendering an UI that allows the user to select an angle from a 360 degrees circle. An angle can be picked by either dragging inside the circle or by entering the angle in a number input.



Props

`AnglePickerControl` accepts the common shared props `onChange`, `value`, and `label`.

Code

```
// Destructure component
const { AnglePickerControl } = wp.components;

// Basic usage
<AnglePickerControl
	label={__('Pick an angle', 'awhitepixel')}
	value={props.attributes.numberAttribute}
	onChange={(val) => props.setAttributes({numberAttribute: val})}
/>
```

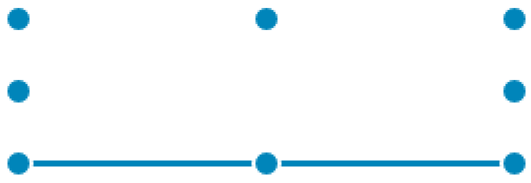
Resize

ResizableBox

wp.components

`ResizableBox` is a component for providing an interface to set the size of something; most commonly the size of the block itself. The user is presented with handles that can be dragged to resize the child element. This component is for example used by WordPress' Cover and Spacer blocks to set the block's minimum height.

The image below shows the handles that appear around the element to resize, with the hover effect over the bottom edge.



Props

`ResizableBox` does not use any of the commonly shared props. The value prop for this component is in the prop `size` and it expects an object with the properties `height` and `width`. The prop `onResizeStop` is the event that occurs when a resizing event has completed, and it returns with four arguments. Common use of this event is adding the `height` and `width` properties of the fourth argument onto the current height and width values. See code example below.

The component supports additional props (see readme documentation), but some worth mentioning are;

`enable`	(Object) (optional) This prop expects an object for controlling which corners and sides to render handles for. For instance WordPress' Spacer block is setting `false` to all corners and edges except the bottom, thus only allowing the user to change the height. Possible properties are `top`, `bottom`, `right`, `left`, `topRight`, `topLeft`,

	`bottomRight`, and `bottomLeft`. For each property you provide either `true` or `false`. See example of use in the code below.
`showHandle`	(Boolean) (optional) Boolean whether or not to render visible handles. Default `false`. I strongly recommend setting this equal to the block's prop `isSelected`. When `showHandle` is `false` it is not possible to resize at all.
`minHeight` `minWidth`	(String) (optional) Set a number (as string) to define the minimum height (`minHeight`) and minimum width (`minWidth`).
`lockAspectRatio`	(Boolean) (optional) If this prop is `true` the aspect ratio will be unchanged. Changing the width or height will also scale the opposite side by an equal amount.

Code

Basic usage allows the user to change all sides and into any size. The basic usage example below assumes the block has two attributes; `boxHeight` and `boxWidth`, both of type `number`.

Note how the new size is saved by adding the height and width properties from the fourth parameter returned from `onResizeStop` onto the existing values. Keep in mind that you should ensure that the values always are valid numbers. If by accident any value gets an invalid number; `NaN`, the resizable box will stop working and get stuck in a loop of updating `NaN` as value every time. The only way to fix it is to delete the block and start over. I strongly recommend providing a default or initial numbers to the attributes to avoid them starting as `undefined`, which will result in `NaN` once the code in `onResizeStop` has been run.

Any child components in the `ResizableBox` component will be the element to resize. It can be the full block's content, an image, or a single HTML node.

The second code example is a more advanced example of a custom block's edit component using `ResizableBox` to define the block's height. It assumes an attribute `boxHeight` of type `number` and `default` 50. In order to get the block to reflect the changed height, the child div-node inside `ResizableBox` gets a `style` property that sets the CSS property `height` to the value of the `boxHeight` attribute. In some cases you might want to consider using `min-height` instead.

The code allows only one handle; the bottom edge. The user cannot change the width with this configuration. Using this setup we can skip providing or updating the width value.

```
// Destructure component
const { ResizableBox } = wp.components;

// Basic usage
<ResizableBox
	size={{ height: props.attributes.boxHeight, width: props.attributes.boxWidth }}
	showHandle={props.isSelected}
	onResizeStop={(event, direction, resize_element, delta) => props.setAttributes({
		boxHeight: parseInt(props.attributes.boxHeight + delta.height),
		boxWidth: parseInt(props.attributes.boxWidth + delta.width)
	})}
>
	<div>..Element to be resized..</div>
</ResizableBox>

// Example of using ResizableBox to adjust a block's height. Code shows full block edit
component:
const BlockEdit = (props) => {
	const blockStyles = {
		height: props.attributes.boxHeight + 'px'
	};

	return(
		<ResizableBox
			size={{ height: props.attributes.boxHeight }}
			showHandle={props.isSslected}
			onResizeStop={(event, direction, resize_element, delta) =>
				props.setAttributes({
				boxHeight: parseInt(props.attributes.boxHeight + delta.height)
			})}
			enable={{
				top: false,
				left: false,
				right: false,
				topLeft: false,
				topRight: false,
				bottom: true,
				bottomLeft: false,
				bottomRight: false
			}}
		>
			<div style={blockStyles}>
				..Content of block..
			</div>
		</ResizableBox>
	);
}
```

Interactive Components

The previous chapter focused on user inputs where components are used to store and update a value of something. This chapter will cover components that are not for storing a value, but are for user interactions and information display. Examples are buttons, modals, notifications, spinners, and wrapper components to streamline the design.

All of the below mentioned components resides in the package `wp.components`.

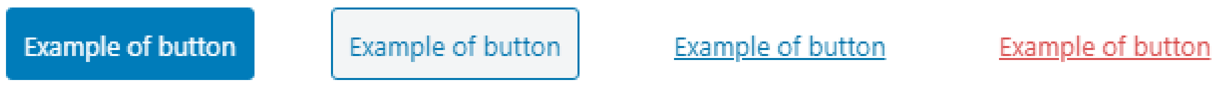
Button

`Button`

`wp.components`

`Button` is a component for rendering a button that allow users to perform an action. The components offer multiple designs controllable via props. What happens when the button is clicked is entirely up to you.

The content of the button is rendered from the child nodes of the Button component. It can be a simple text or more complex HTML. In general there's four main (visible) designs, as shown below:



Primary Secondary Link Destructive link

Props

Passing no props to the component will render a button without any design (no background and no border). It's therefore recommended to pass at least one prop to define its design to make the button look like an actual button.

The component supports additional props (see readme documentation), but some worth mentioning are;

`isPrimary`	(Boolean) (optional) Renders a primary button style (image 1 above).
`isSecondary`	(Boolean) (optional) Renders a secondary button style (image 2 above).
`isLink`	(Boolean) (optional) Renders a button that looks like a link (image 3 above). The button will be stripped of background, border and padding.
`isDestructive`	(Boolean) (optional) Use together with isLink to render a button with red text that indicates destructive behavior (image 4 above). Using this with `isPrimary` or `isSecondary` will unfortunately not make the red text visible.

`isSmall`	(Boolean) (optional) Decreases the size of the button
`isBusy`	(Boolean) (optional) Renders the button with an animated background to indicate that an action is being performed.
`disabled`	(Boolean) (optional) Renders a disabled button that can't be clicked.
`icon`	(String) (optional) Renders an icon before rendering the button's child content. Provide for example a string for a Dashicon, such as `'admin-home'`.
`onClick`	(Function) (optional) Provide a function to run when the button is clicked.

Code

```
// Destructure component
const { Button } = wp.components;

// Basic usage of a primary style button
<Button
	isPrimary
	onClick={ () => console.log('Clicked the button!') }
>
	Example of button
</Button>

// Example of a secondary style button displaying only an icon. In that case we can skip
adding any text inside
<Button
	isSecondary
	icon="admin-home"
	onClick={ () => console.log('Clicked the button!') }
/>
```

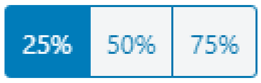
Button Group

`ButtonGroup`

`wp.components`

`ButtonGroup` is a component for helping you render multiple related buttons nicely together. The buttons are displayed next to each other horizontally with no gaps inbetween. Use `ButtonGroup` as a wrapper and render `Button` components as child nodes.

A common use of `ButtonGroup` is showing a small range of small buttons whereas only one is active, like a radio button group. In that case you should clearly indicate which button(s) are active or not. This can be solved by conditionally passing the appropriate design props to the `Button` components.



Props

This component has no props.

Code

In the example below the `ButtonGroup` contains three buttons that each updates the same block attribute `exampleWidth`. By passing button style '`isSecondary`' for all buttons and conditionally add '`isPrimary`' depending on current value, the result will be a button group where only the button for current attribute value is styled as primary.

```
// Destructure the component, along with the Button component
const { ButtonGroup, Button } = wp.components;

<ButtonGroup>
	<Button
		isSecondary
		isPrimary={props.attributes.exampleWidth == '25'}
		onClick={() => props.setAttributes({ exampleWidth: '25' })}
	>25%</Button>
	<Button
		isSecondary
		isPrimary={props.attributes.exampleWidth == '50'}
		onClick={() => props.setAttributes({ exampleWidth: '50' })}
	>50%</Button>
	<Button
		isSecondary
		isPrimary={props.attributes.exampleWidth == '75'}
		onClick={() => props.setAttributes({ exampleWidth: '75' })}
	>75%</Button>
</ButtonGroup>
```

Icon

`Icon, Dashicon`

`wp.components`

Rendering an icon is easy by using either the `Icon` or `Dashicon` components. As the name suggests, `Dashicon` can only be used for displaying one of [WordPress' Dashicons](). The `Icon` component can also be used to render a Dashicon, but also supports providing a custom SVG. Keep in mind that it appears as WordPress is phasing out `Dashicon` in favor of `Icon`.

The icon (Dashicon or SVG) will render a `<svg>` tag without any wrappers. Most commonly you would use this component inside for example a `Button` component.

Props

`icon`	(String | Function | WPComponent) (optional) Provide the icon to render. `Dashicon` supports only a string with the Dashicon's name. For `Icon` you can provide a function or component that returns an `<svg>`. See code examples below.
`size`	(String) (optional) Provide a number as string to define the icon's size. Default is 20 for Dashicon and 24 for other types of icons.
`className`	(String) (optional) If necessary you can provide a desired class name to apply to the `<svg>` tag.

Code

When providing an SVG to `Icon` I recommend to not provide it in a function. As a function the component will not recognize the svg tag and apply the proper size props onto it. It can often result in an unecessary large container. Instead provide the SVG directly.

```
// Destructure either Icon or Dashicon
const { Icon, Dashicon } = wp.components;

// Basic usage of Dashicon
<Dashicon icon="smiley" />

// Basic usage of Icon
<Icon
	icon={
		<svg>
			<circle cx="50%" cy="50%" r="30" fill="red" />
		</svg>
	}
	size="60"
/>

// Not recommended usage: icon={() => (<svg .../>)}
```

Spinner

Spinner

wp.components

The `Spinner` component renders an animated icon that visually informs the user that an action is being performed. This component is commonly rendered in cases where the code needs to wait for an AJAX response or some heavy operation that might take a while.

The spinner looks like the image below, but animated. The white dot is going around in circles.

Props

This component has no props.

Code

Usually you would wrap some conditional around this component. A basic example is a component holding a boolean state for whether or not the spinner should be displayed. When some action is finished the state would be updated to no longer show the spinner.

```
// Destructure component
const { Spinner } = wp.components;

// Basic (and pretty much only) usage
<Spinner />

// Example of rendering a Spinner depending on state
{this.state.isLoading &&
      <Spinner />
}
```

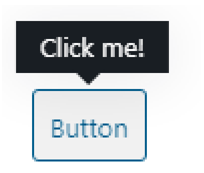
Tooltip

Tooltip

wp.components

If you have an element (e.g. an input or a `Button`) you can wrap it inside a `Tooltip` component to render a tooltip when the element receives focus or on mouseover.

The `Tooltip` component allows only *one* child element. If you need a tooltip to wrap around more advanced content, see "Advanced Tooltip" (`Popover`).



Props

`text`	(String) (optional) The text to display inside the tooltip.
`position`	(String) (optional) Define the direction the tooltip should open relative to its parent node. Possible values are "`top`", "`bottom`" on the y axis and "`left`", "`center`", and "`right`" on the x axis. You can combine the x and y values with a space. Default is "`top center`".

Code

```
// Destructure component
const { Tooltip } = wp.components;

// Basic usage, wrapped around a Button component
<Tooltip
       text="Click me!"
>
       <Button isSecondary>Button</Button>
</Tooltip>
```

Advanced Tooltip

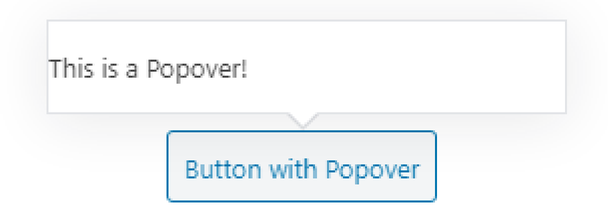
Popover

wp.components

Popover is a more advanced tooltip that works more like a floating modal. It can render content of any sort, not just simple text. As opposed to Tooltip the children elements to this component is what is rendered inside the tooltip. It anchors itself to its parent node (which can for example be a Button).

Another crucial thing to be aware of is that Popover is always rendered visible, and not activated by an element's focus or mouseover (like Tooltip). You will need to handle the render of this component depending on some state or variable. In most cases you might want to consider instead using Dropdown (see "Toggleable Advanced Tooltip") that handles this for you. Dropdown is using Popover to render its content.

Keep in mind that you most likely need to add some custom styling to the Popover's content to make it look good.



Props

The component supports additional props (see readme documentation), but some worth mentioning are;

className	(String) (optional) Provide an additional class name to apply to the popover. This is useful for targeting it with custom styling.
position	(String) (optional) Define the direction the tooltip should open relative to its parent node. Possible values are "top", "bottom" on the y axis and

	"`left`", "`center`", and "`right`" on the x axis. You can combine the x and y values with a space. Default is "`top center`".
`noArrow`	(Boolean) (optional) Set to `true` to hide the arrow that visually indicates the element the `Popover` is anchored to.

Code

The example below is a basic example of a `Popover` conditionally rendered by a state variable, inside a `Button`. Keep in mind that you can add as many child nodes inside the popover as you like. The `Popover`'s size will (mostly) expand to fit its content:

```
// Destructure component
const { Popover } = wp.components;

// Basic usage
<Button isSecondary>
      Button with Popover
      {this.state.visiblePopover && (
            <Popover>
                  <div>
                        <p>This is a Popover!</p>
                  </div>
            </Popover>
      )}
</Button>
```

Toggleable Advanced Tooltip

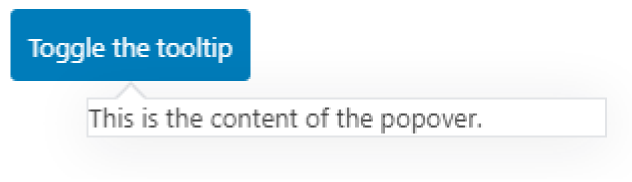
`Dropdown`

`wp.components`

`Dropdown` is a component you can use to render a button to toggle a floating tooltip (using the `Popover` component) when clicked. `Dropdown` takes care of updating the state of the modal (opened/closed) and handles closing the modal when clicking anywhere outside. This is why the `Dropdown` component is a good alternative to manually handling a `Popover` component's render depending on some state and handling the triggers to close the `Popover`.

The component has a prop where you can render the modal toggler, most commonly by rendering a `Button` component. The `Dropdown`'s content is rendered by a prop, not its child nodes.

As with the `Popover` component, you most likely need to add some custom styling to make the `Popover`'s content look good.



Props

The component supports additional props (see readme documentation), but some worth mentioning are;

`renderToggle`	(Function) (required) The function to render the toggler, usually a `Button`. The function receives an object as parameter with the properties: `isOpen`: current open/closed state of the tooltip, `onToggle`: a function to call to toggle the open/closed state, `onClose`: a function to toggle closed state.
`renderContent`	(Function) (required) The function to render the `Popover`'s content. The function receives the same object as parameter as explained in `renderToggle`.

`position`	(String) (optional)) Define the direction the tooltip should open relative to its parent node. Possible values are "`top`", "`bottom`" on the y axis and "`left`", "`center`", and "`right`" on the x axis. You can combine the x and y values with a space. Default is "`top center`".
`className`	(String) (optional) Provide an additional class name to apply to the global container (wrapping around the `Button` and `Popover`).
`contentClassName`	(String) (optional) Provide an additional class name to apply to the `Popover` element.
`popoverProps`	(Object) (optional) If you need to pass over custom props to the `Popover` component that is not exposed in `Dropdown` you can do it in this prop. The prop `noArrow` is an example.
`onToggle`	(Function) (optional) Function to be invoked when the `Popover` changes from open to closed state and vice versa. The function receives a boolean as parameter; if it's `true` the popover is about to open, and if it's `false` the popover will close.

Code

```
// Destructure component, and Button as well
const { Dropdown, Button } = wp.components;

// Basic usage, showing the Popover below and to the right of the toggle button
<Dropdown
      position="bottom right"
      renderToggle={({ onToggle }) => (
            <Button isPrimary onClick={onToggle}>
                  Toggle the tooltip
            </Button>
      )}
      renderContent={() => (
            <div>
                  This is the content of the popover.
            </div>
      )}
/>
```

Dropdown Menu

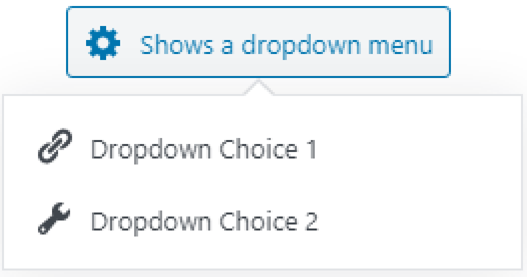
MenuGroup, MenuItem

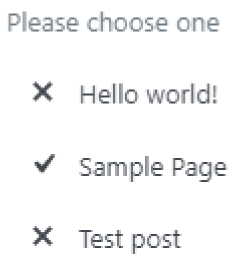
wp.components

If you need to render a list of elements similar to a dropdown menu but don't want to use a standard select; `MenuGroup` and `MenuItem` can be a good alternative. Use `MenuGroup` as the parent node to one or multiple `MenuItem`.

You can use the combination as a dropdown menu, for example to a toolbar button, allowing users to do actions upon clicking the menu elements. Each item can optionally render an icon before the element text. Or you can also use `MenuGroup` and `MenuItem` to display a list of choosable elements, for instance a list of posts to choose from.

The items has a prop to make them work as a checkbox (allows to choose multiple) or as a radio button (only one can be chosen). Keep in mind that they don't visibly render as checkboxes or radio buttons.





Props

The `MenuGroup` component has only one prop: `label`. This is an optional string that will display a label element above the list of `MenuItem`.

As for `MenuItem` there are some more props. It supports additional props (see readme documentation), but some worth mentioning are;

`isSelected`	(Boolean) (optional) Define whether or not the menu item is selected.

`icon`	(String) (optional) Display an icon before the item's text. Provide a Dashicon's name, for example `'yes'` for a checkbox. If you use this component to render a choice of items it can make sense to render a checkbox icon for the selected one(s). See example below.
`role`	(String) (optional) Can be set as either "`menuitemcheckbox`" (to allow multiple `MenuItem` inside a `MenuGroup` to be chosen) or "`menuitemradio`" (only one can be selected). Default is "`menuitemcheckbox`".
`onClick`	(Function) (optional) Function to run when clicking on the menu item.

Code

The first example below is using `MenuGroup` and `MenuItem` to generate a dropdown menu to a button inside a `Popover` (advanced tooltip).

```
// Destructure components
const { MenuGroup, MenuItem } = wp.components;

// First example: As a dropdown menu inside a Popover
<Button isSecondary icon="admin-generic">
	Shows a dropdown menu
	<Popover position="bottom center">
		<MenuGroup>
			<MenuItem
				icon='admin-links'
				onClick={() => console.log('Clicked menu choice 1')}
			>Dropdown Choice 1</MenuItem>
			<MenuItem
				icon='admin-tools'
				onClick={() => console.log('Clicked menu choice 2')}
			>Dropdown Choice 2</MenuItem>
		</MenuGroup>
	</Popover>
</Button>
```

The example below is a basic example of a using `MenuGroup` and `MenuItem` to allow the user to select one element from a list of elements.

It assumes a variable that contains an array of choices and some variable to determine which is the chosen one. It renders a checkbox icon on the selected item and a "x" icon on the others to visually show the user which one is selected:

```
// Second example: As a list of choosable elements
<MenuGroup
	label="Please choose one"
>
	{choices.map((item) => (
		<MenuItem
			isSelected={item == activeChoice}
			icon={item == activeChoice ? 'yes' : 'no-alt'}
			role="menuitemradio"
			onClick={() => console.log('Handle switching selected one')}
		>{item}</MenuItem>
	))}
</MenuGroup>
```

Notice

`Notice`

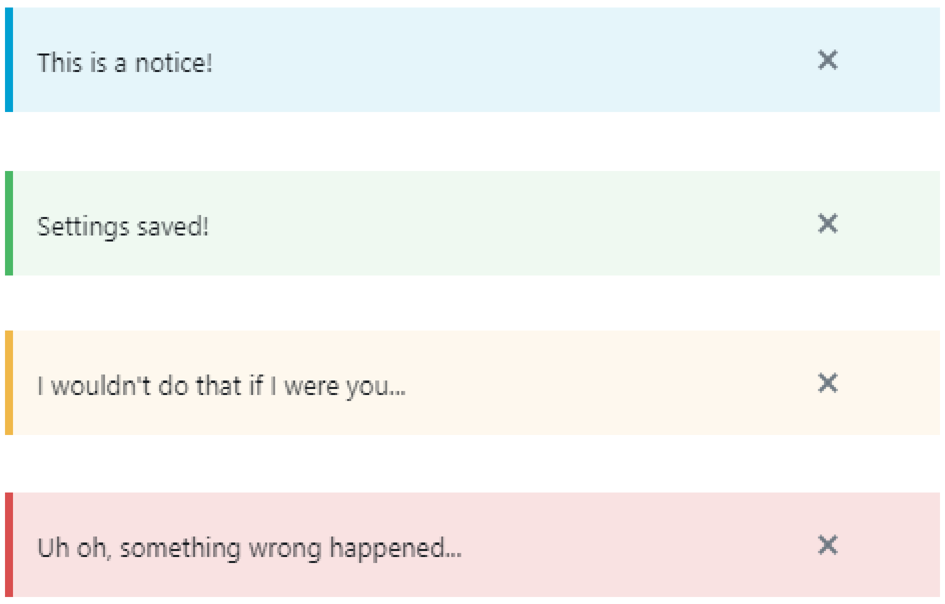
`wp.components`

`Notice` is a component to render clearly visible messages to the user. They should be used for messages that don't necessarily require an action from the user. Common uses are displaying a warning that something went wrong, or showing a success message after an action has been successfully completed. You control the notice's color with props.

The notices are rendered at their position in the code, and you need to write code to render the component depending on a state or after some action.

The children node(s) to the component is what is displayed inside the notice. Just simple text is recommended as children to `Notice`. If you need to display more complex content, see `Modal`.

Notices are dismissable as default, but you can make them indismissable via props if you'd rather control their render by a timer or some other fashion.



This is a notice! ×

Settings saved! ×

I wouldn't do that if I were you... ×

Uh oh, something wrong happened... ×

Props

The component supports additional props (see readme documentation), but some worth mentioning are;

`status`	(String) (optional) Defines the type (design). Can be "`warning`" (yellow), "`success`" (green), "`error`" (red), or "`info`" (blue). Default is "`info`".
`isDismissible`	(Boolean) (optional) Decides whether or not the notice can be closed by the user. Default is `true`. If set to `true` the "X" close button will not appear and the notice will stay until you remove it by some action or state change.

Code

```
// Destructure component
const { Notice } = wp.components;

// Basic usage
<Notice
>
      This is a notice!
</Notice>

// A simple example of a conditional notice that vary the type of message and text
depending on some boolean state wasSuccessful
<Notice
      status={wasSuccessful ? 'success' : 'error'}
>
      {wasSuccessful ? 'Settings saved!' : 'Uh oh, something wrong happened...'}
</Notice>
```

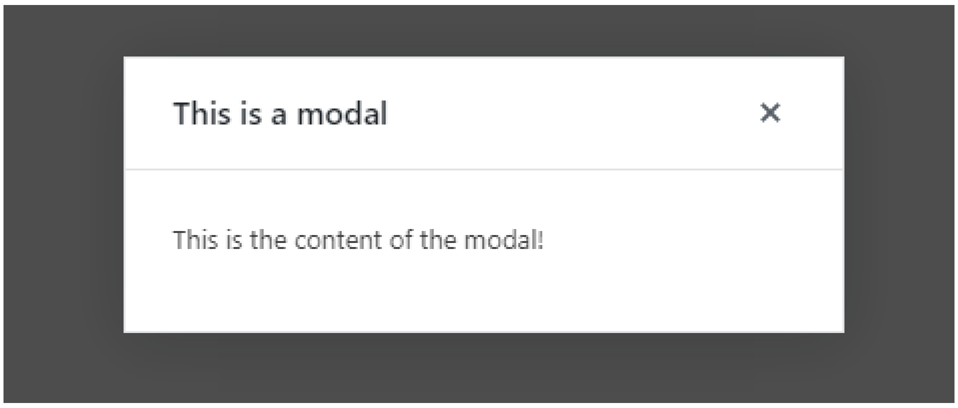
Modal

`Modal`

`wp.components`

`Modal` is a component that renders a floating dialog window. The component renders its children nodes as the content of the modal. The modal fades out the background and will always appear centered in the screen, and it resizes depending on its content. So it doesn't really matter where you add the render code of the modal, but it needs to be conditionally rendered depending on a certain state.

Keep in mind that as default the modal renders with a close button, but nothing happens upon clicking it unless you add props to handle the close event.



Props

The component supports additional props (see readme documentation), but some worth mentioning are;

`title`	(String) (required) Sets the modal's title.
`onRequestClose`	(Function) (required) Function to call to indicate that the modal should be closed. Usually it would set some state to false which prevents the render of the `Modal` component.
`isDismissible`	(Boolean) (optional) Set to `false` to not allow the user to close the modal. The modal is rendered without the "x" close button. You would need to render some kind of button or action inside the modal to allow to close it. Use together with

	`shouldCloseOnEsc` and `shouldCloseOnClickOutside` to make the modal truly impossible to close otherwise.
`shouldCloseOnEsc`	(Boolean) (optional) Set to `false` to prevent closing the modal upon pressing the Escape key. Default is `true`.
`shouldCloseOnClickOutside`	(Boolean) (optional) Set to `false` to prevent closing the modal upon clicking anywhere outside the modal. Default is `true`.
`className`	(String) (optional) Provide a custom class name that will be applied to the modal's content `div`.
`overlayClassName`	(String) (optional) Provide a custom class name to the modal's overlay `div` (the fade out background).

Code

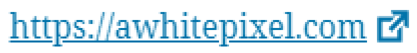
```
// Destructure component
const { Modal } = wp.components;

// Basic example of a Button that opens a modal. It assumes a state isModalOpen which
when true will render the modal
<Button
	isSecondary
	onClick={() => setModalOpen(true)}
>Open modal</Button>
{isModalOpen && (
	<Modal
		title="This is a modal"
		onRequestClose={() => setModalOpen(false)}
	>
		<p>This is the content of the modal!</p>
	</Modal>
)}

// Example of a modal that has disabled all possible close actions unless the user clicks
the manual Button component inside
<Modal
	title="This is a modal"
	isDismissible={false}
	shouldCloseOnEsc={false}
	shouldCloseOnClickOutside={false}
>
	<p>This is the content of the modal!</p>
	<Button isPrimary onClick={() => setModalOpen(false)}>Close modal</Button>
</Modal>
```

Clickable External Link

`ExternalLink`

`wp.components`

`ExternalLink` is a small, simple component that renders a clickable link. It appears with underlined text and the "external link" icon after. The children node is the link text.


https://awhitepixel.com

Props

There's only two props to this component;

`href`	(String) (required) Set the URL the link should go to.
`className`	(String) (optional) Provide a custom class name to the link.

Code

```
// Destructure component
const { ExternalLink } = wp.components;

// Basic usage
<ExternalLink
      href="https://awhitepixel.com"
>
      https://awhitepixel.com
</ExternalLink>
```

Color Preview

`ColorIndicator`

`wp.components`

`ColorIndicator` is a small, simple component that renders a small box previewing one color. You cannot interact with the preview box. It's useful in cases where you want to visually show the user the active or chosen color.

This component is used by the Inspector "Color settings" tab found in most WordPress blocks, right above the palette colors when a custom color is chosen.

Props

There's only one prop to this component;

`colorValue`	(String) (required) Set the hex color (including the "#") to display in the preview box.

Code

```
// Destructure component
const { ColorIndicator } = wp.components;

// Basic usage
<ColorIndicator
	colorValue="#9a6cd8"
/>
```

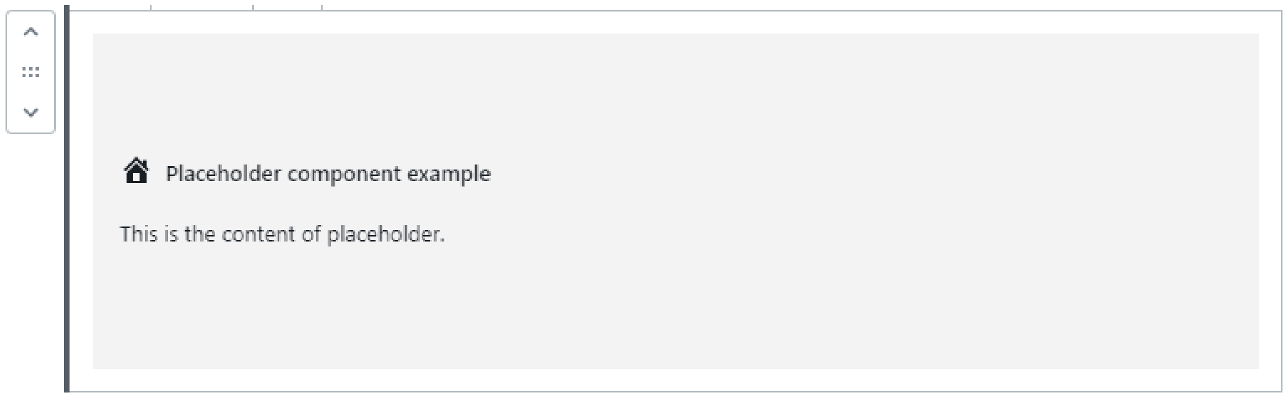
Content wrapper: Placeholder

`Placeholder`

`wp.components`

The `Placeholder` component is used by many blocks inside the block's editor area to signify an "editing" mode. The `Placeholder` generates a flex-styled `div` with light gray transparent background, ready to wrap other components and content in. You can put any type of content inside as this is simply a wrapping div that comes with some styling.

You can see `Placeholder` being used in for example Cover block right after adding it when it displays a choice of choosing a color or setting an image.



Placeholder component example

This is the content of placeholder.

Props

`label`	(String) (optional) Set a label (displayed above the `Placeholder`'s content) in bold.
`icon`	(String | WPElement) (optional) Provide a Dashicon or an SVG icon to display before the label above the `Placeholder`'s content.
`instructions`	(String) (optional) Renders instructional text below the label.
`isColumnLayout`	(Boolean) (optional) Decides the flex-direction of the `Placeholder`'s wrapper. Default is `false`, which means flex-direction is '`column`' (child nodes are placed next to each other horizontally).
`className`	(String) (optional) Add a custom class to the `Placeholder`'s `div`.

Code

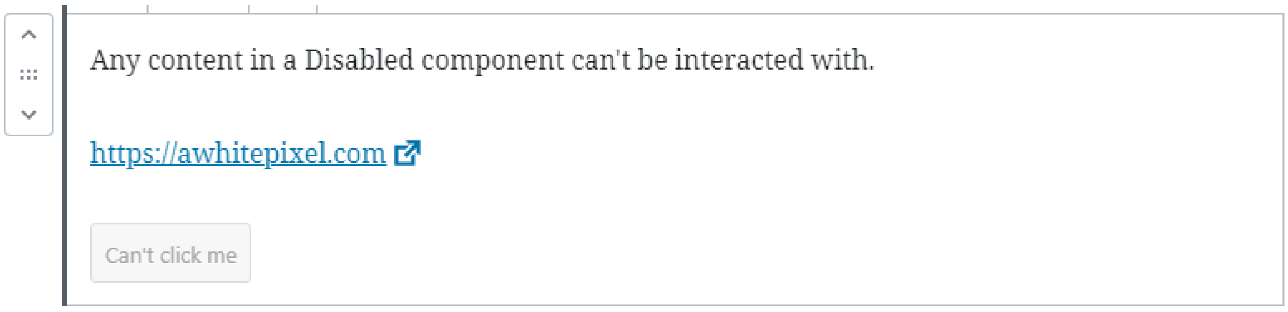
```
// Destructure component
const { Placeholder } = wp.components;

// Basic usage
<Placeholder
      icon="admin-home"
      label="Placeholder component example"
>
      <p>This is the content of placeholder.</p>
</Placeholder>
```

Content wrapper: Disabled

Disabled

wp.components

`Disabled` is a simple wrapper component that disables any interactions with the components inside it. Any input components are uneditable, links are unclickable, and any buttons automatically get disabled. This component is useful when you want to show a preview of a complex block with for example links, but you want to avoid the user accidently clicking the links while in the editor.


Any content in a Disabled component can't be interacted with.

https://awhitepixel.com

Props

`className`	(String) (optional) Add a custom class to the `Disabled`'s `div`.

Code

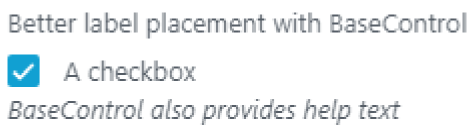
```
// Destructure component
const { Disabled } = wp.components;

// Basic usage
<Disabled>
	<p>Any content in a Disabled component can't be interacted with.</p>
	<Button isSecondary>Can't click me</Button>
</Disabled>
```

Content wrapper: Generic input

`BaseControl`

`wp.components`

The `BaseControl` component is helpful for generating labels and help text for your custom form input. Useful for creating content that follows WordPress standardized design and avoiding adding custom editor styling. Wrap this component around any user input component that doesn't support label or where the label gets misplaced. You can also wrap this around custom HTML form elements in cases you don't want to use WordPress' user input components.


Better label placement with BaseControl

✓ A checkbox

BaseControl also provides help text

Props

`id`	(String) (required) The id to the input element rendered inside the component so that the label and help text are correctly related to the input.
`label`	(String) (optional) The text to be displayed in the label
`help`	(String) (optional) Displays a help text below the content in italic.
`className`	(String) (optional) Add a custom class to the wrapping `div`.

Code

```
// Destructure component
const { BaseControl } = wp.components;

// Example of properly adding a label above a CheckboxControl by using BaseControl
<BaseControl
      label="Better label placement above with BaseControl"
>
      <CheckboxControl label="A checkbox" />
</BaseControl>
```

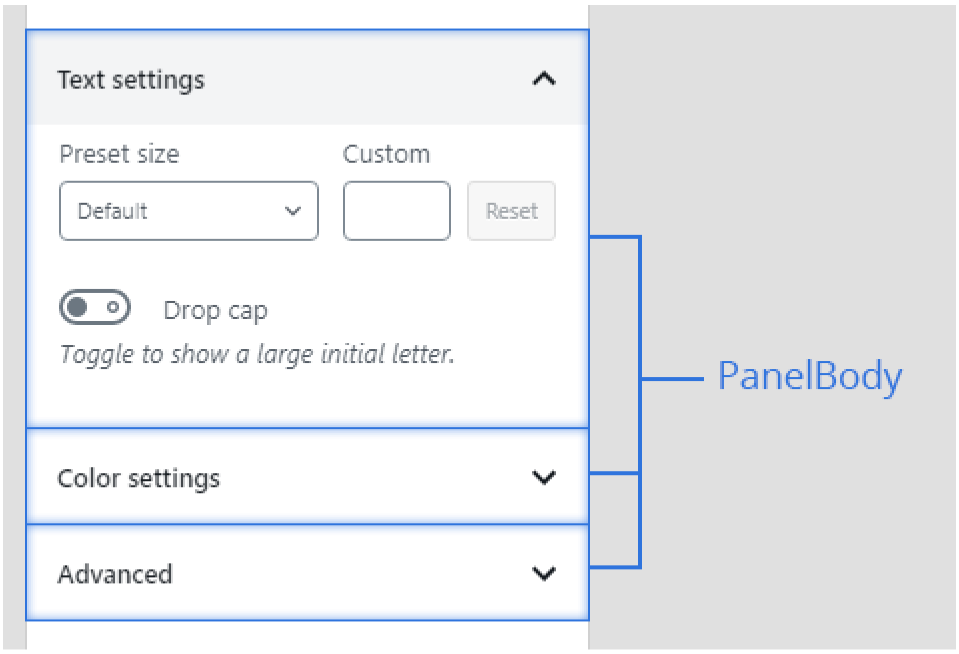
Inspector Section

`PanelBody`

`wp.components`

`PanelBody` is a component that generates a single section of content inside Inspector that can be expanded or collapsed.

This component is meant to be used as child nodes inside the `InspectorControls` component. It is however possible to use it inside block editor content as well (styling and the collapse/expand functionality will work).



Props

The component supports additional props (see readme documentation), but some worth mentioning are;

`title`	(String) (optional) Set the section's title. Not required but important nonetheless as this is the only content visible when a section is collapsed.
`initialOpen`	(Boolean) (optional) Set to `true` to auto-expand the section when rendered.

`opened`	(Boolean) (optional) Set to `true` to keep the section permanently open. Clicking on collapse will do nothing. It will also always be auto expanded, so using `initialOpen` is not necessary.
`icon`	(String) (optional) Provide a Dashicon name and an icon will appear after the section's title.
`className`	(String) (optional) Provide a custom class name to the wrapper `div`.

Code

```
// Destructure component
const { PanelBody } = wp.components;

// Basic usage
<PanelBody
      title="Example of PanelBody"
      initialOpen={true}
>
      Content inside section.
</PanelBody>

// Common usage of adding sections to Inspector
const { InspectorControls } = wp.blockEditor;
const { PanelBody } = wp.components;
...
<InspectorControls>
      <PanelBody
            title="Example of PanelBody"
            initialOpen={true}
      >
            Content inside section.
      </PanelBody>
      <PanelBody
            title="Another PanelBody"
      >
            Content inside another.
      </PanelBody>
</InspectorControls>
```

Inspector Section Content Wrapper

`PanelRow`

`wp.components`

WordPress offers a content wrapper component `PanelRow` to be used inside a `PanelBody` component. `PanelRow` sets its children content as flex with flex-direction `row` (content is placed next to each other horizontally).

Common usage is adding multiple `PanelRow` inside a `PanelBody`, each with a single input component inside.

Props

There's only one prop to `PanelRow`;

`className`	(String) (optional) Provide a custom class name to the wrapping `div`.

Code

```
// Destructure component
const { PanelRow } = wp.components;

// Example of using PanelRow inside InspectorControls (wp.blockEditor) and PanelBody
(wp.components)
<InspectorControls>
      <PanelBody
            title="This is a section"
            initialOpen={true}
      >
            <PanelRow>
                  <TextControl label="Example input" />
            </PanelRow>
            <PanelRow>
                  <CheckboxControl label="Example input" />
            </PanelRow>
      </PanelBody>
</InspectorControls>
```

Final words

I hope this small, free e-book guide has been of some use to you!

This guide was written by *a white pixel*, a WordPress developer living in Norway. I have a website dedicated to self-written and detailed WordPress tutorials at https://awhitepixel.com/. You are most welcome to take a look. There you'll find tutorial and guides for theme development, Gutenberg, general WordPress, and plugin customization for e.g. WooCommerce and Gravity Forms.

- awhitepixel