Edit and Save

- Edit
  - Block wrapper props
  - attributes
  - isSelected
  - setAttributes
- Save
  - block wrapper props
  - attributes
  - innerBlocks
- Examples
  - Saving Attributes to Child Elements
  - Saving Attributes via Serialization
- Validation
  - Validation FAQ

When registering a block with JavaScript on the client, the `edit` and `save` functions provide the interface for how a block is going to be rendered within the editor, how it will operate and be manipulated, and how it will be saved.

Edit

The `edit` function describes the structure of your block in the context of the editor. This represents what the editor will render when the block is used.

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


// ...
const blockSettings = {
    apiVersion: 3,
```

```
    // ...

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

        return <div { ...blockProps }>Your block.</div>;
    },
};
```

Block wrapper props

The first thing to notice here is the use of the `useBlockProps` React hook on the block wrapper element. In the example above, the block wrapper renders a “div” in the editor, but in order for the Gutenberg editor to know how to manipulate the block, add any extra classNames that are needed for the block... the block wrapper element should apply props retrieved from the `useBlockProps` react hook call. The block wrapper element should be a native DOM element, like `<div>` and `<table>`, or a React component that forwards any additional props to native DOM elements. Using a `<Fragment>` or `<ServerSideRender>` component, for instance, would be invalid.

If the element wrapper needs any extra custom HTML attributes, these need to be passed as an argument to the `useBlockProps` hook. For example to add a `my-random-classname` className to the wrapper, you can use the following code:

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

// ...
const blockSettings = {
    apiVersion: 3,

    // ...

    edit: () => {
        const blockProps = useBlockProps( {
            className: 'my-random-classname',
        } );

        return <div { ...blockProps }>Your block.</div>;
    },
};
```

attributes

The `edit` function also receives a number of properties through an object argument. You can use these properties to adapt the behavior of your block.

The `attributes` property surfaces all the available attributes and their corresponding values, as described by the `attributes` property when the block type was registered. See attributes documentation for how to specify attribute sources.

In this case, assuming we had defined an attribute of `content` during block registration, we would receive and use that value in our edit function:

```
edit: ( { attributes } ) => {
    const blockProps = useBlockProps();

    return <div { ...blockProps }>{ attributes.content }</div>;
};
```

The value of `attributes.content` will be displayed inside the `div` when inserting the block in the editor.

isSelected

The isSelected property is a boolean that communicates whether the block is currently selected.

```
edit: ( { attributes, isSelected } ) => {
    const blockProps = useBlockProps();

    return (
        <div { ...blockProps }>
            Your block.
            { isSelected && (
                <span>Shows only when the block is selected.</span>
            ) }
        </div>
    );
};
```

setAttributes

This function allows the block to update individual attributes based on user interactions.

```
edit: ( { attributes, setAttributes, isSelected } ) => {
    const blockProps = useBlockProps();

    // Simplify access to attributes
    const { content, mySetting } = attributes;

    // Toggle a setting when the user clicks the button
    const toggleSetting = () => setAttributes( { mySetting: ! mySetting } );
    return (
        <div { ...blockProps }>
            { content }
            { isSelected && (
                <button onClick={ toggleSetting }>Toggle setting</button>
            ) }
        </div>
    );
};
```

When using attributes that are objects or arrays it's a good idea to copy or clone the attribute prior to updating it:

```
// Good - a new array is created from the old list attribute and a new list item:
const { list } = attributes;
const addListItem = ( newListItem ) =>
    setAttributes( { list: [ ...list, newListItem ] } );

// Bad - the list from the existing attribute is modified directly to add the new list item:
const { list } = attributes;
const addListItem = ( newListItem ) => {
    list.push( newListItem );
    setAttributes( { list } );
};
```

Why do this? In JavaScript, arrays and objects are passed by reference, so this practice ensures changes won't affect other code that might hold references to the same data. Furthermore, the Gutenberg project follows the philosophy of the Redux library that state should be immutable—data should not be changed directly, but instead a new version of the data created containing the changes.

The `setAttributes` also supports an updater function as an argument. It must be a pure function, which takes current attributes as its only argument and returns updated attributes. This method is helpful when you want to update a value based on a previous state or when working with objects and arrays.

Note: *Since WordPress 6.9.*

```
// Toggle a setting when the user clicks the button.
const toggleSetting = () =>
    setAttributes( ( currentAttr ) => ( {
        mySetting: ! currentAttr.mySetting,
    } ) );

// Add item to the list.
const addListItem = ( newListItem ) =>
    setAttributes( ( currentAttr ) => ( {
        list: [ ...currentAttr.list, newListItem ],
    } ) );
```

Save

The `save` function defines the way in which the different attributes should be combined into the final markup, which is then serialized into `post_content`.

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

    return <div { ...blockProps }> Your block. </div>;
};
```

For most blocks, the return value of `save` should be an instance of WordPress Element representing how the block is to appear on the front of the site.

Note: While it is possible to return a string value from `save`, it *will be escaped*. If the string includes HTML markup, the markup will be shown on the front of the site verbatim, not as the equivalent HTML node content. If you must return raw HTML from `save`, use `wp.element.RawHTML`. As the name implies, this is prone to cross-site scripting and therefore is discouraged in favor of a WordPress Element hierarchy whenever possible.

Note: The save function should be a pure and stateless function that depends only on the attributes used to invoke it. It shouldn't use any APIs such as `useState` or `useEffect`, nor retrieve information from another source; for example, it is not possible to use the data module inside – `select( store ).selector( ... )`.
This is because if the external information changes, the block may be flagged as invalid when the post is later edited (read more about Validation).

If there is a need to have other information as part of the save, developers can consider one of these two alternatives:

- Use dynamic blocks and dynamically retrieve the required information on the server.
- Store the external value as an attribute which is dynamically updated in the block's `edit` function as changes occur.

For dynamic blocks, the return value of `save` could represent a cached copy of the block's content to be shown only in case the plugin implementing the block is ever disabled.

If left unspecified, the default implementation will save no markup in post content for the dynamic block, instead deferring this to always be calculated when the block is shown on the front of the site.

block wrapper props

Like the `edit` function, when rendering static blocks, it's important to add the block props returned by `useBlockProps.save()` to the wrapper element of your block. This ensures that the block class name is rendered properly in addition to any HTML attribute injected by the block supports API.

attributes

As with `edit`, the `save` function also receives an object argument including attributes which can be inserted into the markup.

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

    return <div { ...blockProps }>{ attributes.content }</div>;
};
```

When saving your block, you want to save the attributes in the same format specified by the attribute source definition. If no attribute source is specified, the attribute will be saved to the block's comment delimiter. See the Block Attributes documentation for more details.

innerBlocks

There is a second property in the props passed to the `save` function, `innerBlocks`. This property is typically used for internal operations, and there are very few scenarios where you would need to use it.

`innerBlocks`, when initialized, is an array containing object representations of nested blocks. In those rare cases where you might use this property,
it can help you adjust how a block is rendered. For example, you could render a block differently based on the number of nested blocks or if a specific block type is present..

```
save: ( { attributes, innerBlocks } ) => {
    let { className, ...rest } = useBlockProps.save();

    // innerBlocks could also be an object - react element during initialization
    const numberOfInnerBlocks = innerBlocks?.length;
    if ( numberOfInnerBlocks > 1 ) {
        className = className + ( className ? ' ' : '' ) + 'more-than-one';
    };
    const blockProps =  { ...rest, className };

    return <div { ...blockProps }>{ attributes.content }</div>;
};
```

Here, an additional class is added to the block if number of inner blocks is greater than one, allowing for different styling of the block.

Examples

Here are a couple examples of using attributes, edit, and save all together.

Saving Attributes to Child Elements

```
attributes: {
    content: {
        type: 'string',
        source: 'html',
        selector: 'div'
```

```
    }
},

edit: ( { attributes, setAttributes } ) => {
    const blockProps = useBlockProps();
    const updateFieldValue = ( val ) => {
        setAttributes( { content: val } );
    }
    return (
        <div { ...blockProps }>
            <TextControl
                label='My Text Field'
                value={ attributes.content }
                onChange={ updateFieldValue }
            />
        </div>
    );
},

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

    return <div { ...blockProps }> { attributes.content } </div>;
},
```

Saving Attributes via Serialization

Ideally, the attributes saved should be included in the markup. However, there are times when this is not practical, so if no attribute source is specified the attribute is serialized and saved to the block's comment delimiter.

This example could be for a dynamic block, such as the Latest Posts block, which renders the markup server-side. The save function is still required, however in this case it simply returns null since the block is not saving content from the editor.

```
attributes: {
    postsToShow: {
        type: 'number',
    }
```

```
},

edit: ( { attributes, setAttributes } ) => {
    const blockProps = useBlockProps();

    return (
        <div { ...blockProps }>
            <TextControl
                label='Number Posts to Show'
                value={ attributes.postsToShow }
                onChange={ ( val ) => {
                    setAttributes( { postsToShow: parseInt( val ) } );
                }}
            />
        </div>
    );
},

save: () => {
    return null;
}
```

Validation

When the editor loads, all blocks within post content are validated to determine their accuracy in order to protect against content loss. This is closely related to the saving implementation of a block, as a user may unintentionally remove or modify their content if the editor is unable to restore a block correctly. During editor initialization, the saved markup for each block is regenerated using the attributes that were parsed from the post's content. If the newly-generated markup does not match what was already stored in post content, the block is marked as invalid. This is because we assume that unless the user makes edits, the markup should remain identical to the saved content.

If a block is detected to be invalid, the user will be prompted to choose how to handle the invalidation:

Invalid block prompt

Clicking the **Attempt Block Recovery** button will attempt a recovery action as much as possible.

Clicking the "3-dot" menu on the side of the block displays three options:

- **Resolve**: Open Resolve Block dialog box with two buttons:
  - **Convert to HTML**: Protects the original markup from the saved post content and converts the block from its original type to the HTML block type, enabling the user to modify the HTML markup directly.
  - **Convert to Blocks**: Protects the original markup from the saved post content and converts the block from its original type to the validated block type.
- **Convert to HTML**: Protects the original markup from the saved post content and converts the block from its original type to the HTML block type, enabling the user to modify the HTML markup directly.
- **Convert to Classic Block**: Protects the original markup from the saved post content as correct. Since the block will be converted from its original type to the Classic block type, it will no longer be possible to edit the content using controls available for the original block type.

Validation FAQ

How do blocks become invalid?

The two most common sources of block invalidations are:

1. A flaw in a block's code would result in unintended content modifications. See the question below on how to debug block invalidation as a plugin author.
2. You or an external editor changed the HTML markup of the block in such a way that it is no longer considered correct.

I'm a plugin author. What should I do to debug why my blocks are being marked as invalid?

Before starting to debug, be sure to familiarize yourself with the validation step described above documenting the process for detecting whether a block is invalid. A block is invalid if its regenerated markup does not match what is saved in post content, so often this can be caused by the attributes of a block being parsed incorrectly from the saved content.

If you're using attribute sources, be sure that attributes sourced from markup are saved exactly as you expect, and in the correct type (usually a `'string'` or `'number'`).

When a block is detected as invalid, a warning will be logged into your browser's developer tools console. The warning will include specific details about the exact point at which a difference in markup occurred. Be sure to look closely at any differences in the expected and actual markups to see where problems are occurring.

I've changed my block's `save` behavior and old content now includes invalid blocks. How can I fix this?

Refer to the guide on Deprecated Blocks to learn more about how to accommodate legacy content in intentional markup changes.

□□□ #2
□ □□□ □□□ 2026-10-31 18:10:44 UTC
□ □□□ □□□ 2026-10-31 18:10:46 UTC