# Block Filters


- Registration
  - block_type_metadata
  - block_type_metadata_settings
  - register_block_type_args
  - blocks.registerBlockType
- Front end
  - render_block
  - render_block_{namespace/block}
- Editor
  - blocks.getSaveElement
  - blocks.getSaveContent.extraProps
  - blocks.getBlockDefaultClassName
  - blocks.switchToBlockType.transformedBlock
  - blocks.getBlockAttributes
  - editor.BlockEdit
  - editor.BlockListBlock
  - editor.postContentBlockTypes
- Removing Blocks
  - Using a deny list
  - Using an allow list
- Hiding blocks from the inserter
  - allowed_block_types_all
- Managing block categories
  - block_categories_all
  - wp.blocks.updateCategory


WordPress exposes several APIs that allow you to modify the behavior of existing blocks.

# Registration

Blocks in WordPress are typically registered on both the server and client side using `block.json` metadata. You can use the following filters to modify or extend block settings during their registration on the server with PHP and on the client with JavaScript. To learn more, refer to the block registration guide.

## `block_type_metadata`

Filters the raw metadata loaded from the `block.json` file when registering a block type on the server with PHP. It allows modifications to be applied before the metadata gets processed.

The callback function for this filter receives one parameter:

- `$metadata` (`array`): Metadata loaded from `block.json` for registering a block type.

The following example sets the `apiVersion` of all blocks to `2`.

```
function example_filter_metadata_registration( $metadata ) {
    $metadata['apiVersion'] = 2;
    return $metadata;
};
add_filter( 'block_type_metadata', 'example_filter_metadata_registration' );
```

Here's a more robust example that disables background color and gradient support for Heading blocks. The `block_type_metadata` filter is excellent for curating the Editor experience.

```
function example_disable_heading_background_color_and_gradients( $metadata ) {

    // Only apply the filter to Heading blocks.
    if ( ! isset( $metadata['name'] ) || 'core/heading' !== $metadata['name'] ) {
        return $metadata;
    }

    // Check if 'supports' key exists.
    if ( isset( $metadata['supports'] ) && isset( $metadata['supports']['color'] ) ) {

        // Remove Background color and Gradients support.
        $metadata['supports']['color']['background'] = false;
        $metadata['supports']['color']['gradients']  = false;
```

```
    }

    return $metadata;
}
add_filter( 'block_type_metadata', 'example_disable_heading_background_color_and_gradients' );
```

## `block_type_metadata_settings`

Filters the settings determined from the processed block type metadata. It makes it possible to apply custom modifications using the block metadata that isn't handled by default.

The callback function for this filter receives two parameters:

- `$settings` (`array`): Array of determined settings for registering a block type.
- `$metadata` (`array`): Metadata loaded from the `block.json` file.

The following example increases the `apiVersion` for all blocks by `1`.

```
function example_filter_metadata_registration( $settings, $metadata ) {
    $settings['api_version'] = $metadata['apiVersion'] + 1;
    return $settings;
};
add_filter( 'block_type_metadata_settings', 'example_filter_metadata_registration', 10, 2 );
```

## `register_block_type_args`

Filters a block's arguments array (`$args`) right before the block type is officially registered on the server.

The callback function for this filter receives two parameters:

- `$args` (`array`): Array of arguments for registering a block type.
- `$block_type` (`string`): Block type name including namespace.

`register_block_type_args` is the most low-level PHP filter available, and it will work for every block registered on the server. All settings defined on the server are propagated to the client with higher priority than those set in the client.

The following code will disable the color controls for Paragraph, Heading, List, and List Item blocks.

```
function example_disable_color_for_specific_blocks( $args, $block_type ) {
```

```
    // List of block types to modify.
    $block_types_to_modify = [
        'core/paragraph',
        'core/heading',
        'core/list',
        'core/list-item'
    ];

    // Check if the current block type is in the list.
    if ( in_array( $block_type, $block_types_to_modify, true ) ) {
        // Disable color controls.
        $args['supports']['color'] = array(
            'text'       => false,
            'background' => false,
            'link'       => false,
        );
    }

    return $args;
}
add_filter( 'register_block_type_args', 'example_disable_color_for_specific_blocks', 10, 2 );
```

## `blocks.registerBlockType`

Used to filter the block settings when registering the block on the client with JavaScript. It receives the block settings, the name of the registered block, and either null or the deprecated block settings (when applied to a registered deprecation) as arguments. This filter is also applied to each of a block's deprecated settings.

The following example ensures that List blocks are saved with the canonical generated class name (`wp-block-list`):

```
function addListBlockClassName( settings, name ) {
    if ( name !== 'core/list' ) {
        return settings;
    }

    return {
        ...settings,
        supports: {
```

```
            ...settings.supports,
            className: true,
        },
    };
}

wp.hooks.addFilter(
    'blocks.registerBlockType',
    'my-plugin/class-names/list-block',
    addListBlockClassName
);
```

# Front end

The following PHP filters are available to change the output of a block on the front end.

## `render_block`

Filters the front-end content of any block. This filter has no impact on the behavior of blocks in the Editor.

The callback function for this filter receives three parameters:

- `$block_content` (`string`): The block content.
- `$block` (`array`): The full block, including name and attributes.
- `$instance` (`WP_Block`): The block instance.

In the following example, the class `example-class` is added to all Paragraph blocks on the front end. Here the HTML API is used to easily add the class instead of relying on regex.

```
function example_add_custom_class_to_paragraph_block( $block_content, $block ) {

    // Check if the block is a Paragraph block.
    if ( 'core/paragraph' === $block['blockName'] ) {

        // Add the custom class to the block content using the HTML API.
        $processor = new WP_HTML_Tag_Processor( $block_content );

        if ( $processor->next_tag( 'p' ) ) {
```

```
            $processor->add_class( 'example-class' );
        }

        return $processor->get_updated_html();
    }

    return $block_content;
}
add_filter( 'render_block', 'example_add_custom_class_to_paragraph_block', 10, 2 );
```

## `render_block_{namespace/block}`

Filters the front-end content of the defined block. This is just a simpler form of `render_block` when you only need to modify a specific block type.

The callback function for this filter receives three parameters:

- `$block_content` (`string`): The block content.
- `$block` (`array`): The full block, including name and attributes.
- `$instance` (`WP_Block`): The block instance.

In the following example, the class `example-class` is added to all Paragraph blocks on the front end. Notice that compared to the `render_block` example above, you no longer need to check the block type before modifying the content. Again, the HTML API is used instead of regex.

```
function example_add_custom_class_to_paragraph_block( $block_content, $block ) {

    // Add the custom class to the block content using the HTML API.
    $processor = new WP_HTML_Tag_Processor( $block_content );

    if ( $processor->next_tag( 'p' ) ) {
        $processor->add_class( 'example-class' );
    }

    return $processor->get_updated_html();
}
add_filter( 'render_block_core/paragraph', 'example_add_custom_class_to_paragraph_block', 10, 2 );
```

# Editor

The following JavaScript filters are available to change the behavior of blocks while editing in the Editor.

## `blocks.getSaveElement`

A filter that applies to the result of a block's `save` function. This filter is used to replace or extend the element, for example using `React.cloneElement` to modify the element's props, replace its children, or return an entirely new element.

The callback function for this filter receives three parameters:

- `element` (`Object`): The element to be modified and returned.
- `blockType` (`Object`): A block-type definition object.
- `attributes` (`Object`): The block's attributes.

The following example wraps a Cover block in an outer container `div`.

```
function wrapCoverBlockInContainer( element, blockType, attributes ) {

    // Skip if element is undefined.
    if ( ! element ) {
        return;
    }

    // Only apply to Cover blocks.
    if ( blockType.name !== 'core/cover' ) {
        return element;
    }

    // Return the element wrapped in a div.
    return <div className="cover-block-wrapper">{ element }</div>;
}

wp.hooks.addFilter(
    'blocks.getSaveElement',
    'my-plugin/wrap-cover-block-in-container',
    wrapCoverBlockInContainer
);
```

## `blocks.getSaveContent.extraProps`

A filter that applies to all blocks returning a WP Element in the `save` function. This filter is used to add extra props to the root element of the `save` function. For example, you could add a className, an id, or any valid prop for this element.

The callback function for this filter receives three parameters:

- `props` (`Object`): The current `save` element's props to be modified and returned.
- `blockType` (`Object`): A block-type definition object.
- `attributes` (`Object`): The block's attributes.

The following example adds a red background by default to all blocks.

```
function addBackgroundColorStyle( props ) {
    return {
        ...props,
        style: { backgroundColor: 'red' },
    };
}

wp.hooks.addFilter(
    'blocks.getSaveContent.extraProps',
    'my-plugin/add-background-color-style',
    addBackgroundColorStyle
);
```

*Note:* A block validation error will occur if this filter modifies existing content the next time the post is edited. The Editor verifies that the content stored in the post matches the content output by the `save()` function.

To avoid this validation error, use `render_block` server-side to modify existing post content instead of this filter. See render_block documentation.

## `blocks.getBlockDefaultClassName`

Generated HTML classes for blocks follow the `wp-block-{name}` nomenclature. This filter allows to provide an alternative class name.

```
// Our filter function.
function setBlockCustomClassName( className, blockName ) {
```

```
    return blockName === 'core/code' ? 'my-plugin-code' : className;
}

// Adding the filter.
wp.hooks.addFilter(
    'blocks.getBlockDefaultClassName',
    'my-plugin/set-block-custom-class-name',
    setBlockCustomClassName
);
```

## `blocks.switchToBlockType.transformedBlock`

Used to filter an individual transform result from block transformation. All of the original blocks are passed since transformations are many-to-many, not one-to-one.

## `blocks.getBlockAttributes`

Called immediately after the default parsing of a block's attributes and before validation to allow a plugin to manipulate attribute values in time for validation and/or the initial values rendering of the block in the editor.

The callback function for this filter accepts 4 parameters:
– `blockAttributes` (`Object`): All block attributes.
– `blockType` (`Object`): The block type.
– `innerHTML` (`string`): Raw block content.
– `attributes` (`object`): Known block attributes (from delimiters).

In the example below, we use the `blocks.getBlockAttributes` filter to lock the position of all paragraph blocks on a page.

```
// Our filter function
function lockParagraphs( blockAttributes, blockType, innerHTML, attributes  ) {
    if('core/paragraph' === blockType.name) {
        blockAttributes['lock'] = {move: true}
    }
    return blockAttributes;
}

// Add the filter
wp.hooks.addFilter(
    'blocks.getBlockAttributes',
```

```
    'my-plugin/lock-paragraphs',
    lockParagraphs
);
```

## `editor.BlockEdit`

Used to modify the block's `edit` component. It receives the original block `BlockEdit` component and returns a new wrapped component.

The following example adds a new Inspector panel for all blocks.

```
const { createHigherOrderComponent } = wp.compose;
const { InspectorControls } = wp.blockEditor;
const { PanelBody } = wp.components;

const withMyPluginControls = createHigherOrderComponent( ( BlockEdit ) => {
    return ( props ) => {
        return (
            <>
                <BlockEdit key="edit" { ...props } />
                <InspectorControls>
                    <PanelBody>My custom control</PanelBody>
                </InspectorControls>
            </>
        );
    };
}, 'withMyPluginControls' );

wp.hooks.addFilter(
    'editor.BlockEdit',
    'my-plugin/with-inspector-controls',
    withMyPluginControls
);
```

Note that as this hook is run for *all blocks*, consuming it has the potential for performance regressions, particularly around block selection metrics.

To mitigate this, consider whether any work you perform can be altered to run only under certain conditions.

For example, suppose you are adding components that only need to render when the block is *selected*. In that case, you can use the block's "selected" state (`props.isSelected`) to conditionalize your rendering.

The following example adds a new Inspector panel for all blocks, but only when a block is selected.

```
const withMyPluginControls = createHigherOrderComponent( ( BlockEdit ) => {
    return ( props ) => {
        return (
            <>
                <BlockEdit { ...props } />
                { props.isSelected && (
                    <InspectorControls>
                        <PanelBody>My custom control</PanelBody>
                    </InspectorControls>
                ) }
            </>
        );
    };
}, 'withMyPluginControls' );
```

## `editor.BlockListBlock`

Used to modify the block's wrapper component containing the block's `edit` component and all toolbars. It receives the original `BlockListBlock` component and returns a new wrapped component.

The following example adds a unique class name to all blocks.

```
const { createHigherOrderComponent } = wp.compose;

const withClientIdClassName = createHigherOrderComponent(
    ( BlockListBlock ) => {
        return ( props ) => {
            return (
                <BlockListBlock
                    { ...props }
                    className={ 'block-' + props.clientId }
                />
            );
        };
```

```
    },
    'withClientIdClassName'
);

wp.hooks.addFilter(
    'editor.BlockListBlock',
    'my-plugin/with-client-id-class-name',
    withClientIdClassName
);
```

You can add new properties to the block's wrapper component using the `wrapperProps` property of the returned component as shown in the following example.

```
const { createHigherOrderComponent } = wp.compose;
const withMyWrapperProp = createHigherOrderComponent( ( BlockListBlock ) => {
    return ( props ) => {
        const wrapperProps = {
            ...props.wrapperProps,
            'data-my-property': 'the-value',
        };
        return <BlockListBlock { ...props } wrapperProps={ wrapperProps } />;
    };
}, 'withMyWrapperProp' );

wp.hooks.addFilter(
    'editor.BlockListBlock',
    'my-plugin/with-my-wrapper-prop',
    withMyWrapperProp
);
```

## `editor.postContentBlockTypes`

Used to modify the list of blocks that should be enabled even when used inside a locked template. Any block that saves data to a post should be added here. An example of this is the Post Featured Image block. Often used in templates, this block should still allow selecting the image even when the template is locked.

The following example enables the fictitious block `namespace/example`.

```
const addExampleBlockToPostContentBlockTypes = ( blockTypes ) => {
    return [ ...blockTypes, 'namespace/example' ];
};

wp.hooks.addFilter(
    'editor.postContentBlockTypes',
    'my-plugin/post-content-block-types',
    addExampleBlockToPostContentBlockTypes
);
```

# Removing Blocks

## Using a deny list

Adding blocks is easy enough, and removing them is as easy. Plugin or theme authors can “unregister” blocks using a deny list in JavaScript.

Place the following code in a `my-plugin.js` file.

```
// my-plugin.js
import { unregisterBlockType } from '@wordpress/blocks';
import domReady from '@wordpress/dom-ready';

domReady( function () {
    unregisterBlockType( 'core/verse' );
} );
```

Then, load this script in the Editor using the following function.

```
<?php
// my-plugin.php

function my_plugin_deny_list_blocks() {
    wp_enqueue_script(
        'my-plugin-deny-list-blocks',
        plugins_url( 'my-plugin.js', __FILE__ ),
        array( 'wp-blocks', 'wp-dom-ready', 'wp-edit-post' )
    );
```

```
}
add_action( 'enqueue_block_editor_assets', 'my_plugin_deny_list_blocks' );
```

> When unregistering a block, there can be a race condition on which code runs first: registering the block or unregistering the block. You want your unregister code to run last. To do this, you must specify the component that is registering the block as a dependency, in this case, `wp-edit-post`. Additionally, using `wp.domReady()` ensures the unregister code runs once the dom is loaded.

# Using an allow list

If you want to disable all blocks except an allow list, you can adapt the script above like so:

```
// my-plugin.js

var allowedBlocks = [
    'core/paragraph',
    'core/image',
    'core/html',
    'core/freeform',
];

wp.blocks.getBlockTypes().forEach( function ( blockType ) {
    if ( allowedBlocks.indexOf( blockType.name ) === -1 ) {
        wp.blocks.unregisterBlockType( blockType.name );
    }
} );
```

# Hiding blocks from the inserter

## `allowed_block_types_all`

> Before WordPress 5.8, this hook was known as `allowed_block_types`, which is now deprecated. If you need to support older versions of WordPress, you might need a way to detect which filter should be used. You can check if `allowed_block_types` is safe to use by

> seeing if the `WP_Block_Editor_Context` class exists, which was introduced in 5.8.

On the server, you can filter the list of blocks shown in the inserter using the `allowed_block_types_all` filter. You can return either true (all block types supported), false (no block types supported), or an array of block type names to allow. You can also use the second provided parameter `$editor_context` to filter block types based on their content.

```
<?php
// my-plugin.php
function example_filter_allowed_block_types_when_post_provided( $allowed_block_types,
$editor_context ) {
    if ( ! empty( $editor_context->post ) ) {
        return array( 'core/paragraph', 'core/heading' );
    }
    return $allowed_block_types;
}
add_filter( 'allowed_block_types_all',
'example_filter_allowed_block_types_when_post_provided', 10, 2 );
```

# Managing block categories

## `block_categories_all`

> Before WordPress 5.8, this hook was known as `block_categories`, which is now deprecated. If you need to support older versions of WordPress, you might need a way to detect which filter should be used. You can check if `block_categories` is safe to use by seeing if the `WP_Block_Editor_Context` class exists, which was introduced in 5.8.

It is possible to filter the list of default block categories using the `block_categories_all` filter. You can do it on the server by implementing a function which returns a list of categories. It is going to be used during block registration and to group blocks in the inserter. You can also use the second provided parameter `$editor_context` to filter the based on its content.

```
// my-plugin.php
function example_filter_block_categories_when_post_provided( $block_categories,
$editor_context ) {
    if ( ! empty( $editor_context->post ) ) {
```

```
        array_push(
            $block_categories,
            array(
                'slug'  => 'custom-category',
                'title' => __( 'Custom Category', 'custom-plugin' ),
                'icon'  => null,
            )
        );
    }
    return $block_categories;
}
add_filter( 'block_categories_all', 'example_filter_block_categories_when_post_provided', 10,
2 );
```

## `wp.blocks.updateCategory`

You can also display an icon with your block category by setting an `icon` attribute. The value can be the slug of a WordPress Dashicon.

You can also set a custom icon in SVG format. To do so, the icon should be rendered and set on the frontend, so it can make use of WordPress SVG, allowing mobile compatibility and making the icon more accessible.

To set an SVG icon for the category shown in the previous example, add the following example JavaScript code to the Editor calling `wp.blocks.updateCategory` e.g:

```
( function () {
    var el = React.createElement;
    var SVG = wp.primitives.SVG;
    var circle = el( 'circle', {
        cx: 10,
        cy: 10,
        r: 10,
        fill: 'red',
        stroke: 'blue',
        strokeWidth: '10',
    } );
    var svgIcon = el(
        SVG,
        { width: 20, height: 20, viewBox: '0 0 20 20' },
        circle
```

```
    );
    wp.blocks.updateCategory( 'my-category', { icon: svgIcon } );
} )();
```

---

□□□ #2
□ □□□ □□□ 2026-10-31 18:11:31 UTC
□ □□□ □□□ 2026-10-31 18:11:33 UTC