@wordpress/blocks

- Installation
- API
  - cloneBlock
  - cloneSanitizedBlock
  - createBlock
  - createBlocksFromInnerBlocksTemplate
  - doBlocksMatchTemplate
  - findTransform
  - getBlockAttributes
  - getBlockAttributesNamesByRole
  - getBlockBindingsSource
  - getBlockBindingsSources
  - getBlockContent
  - getBlockDefaultClassName
  - getBlockFromExample
  - getBlockMenuDefaultClassName
  - getBlockSupport
  - getBlockTransforms
  - getBlockType
  - getBlockTypes
  - getChildBlockNames
  - getDefaultBlockName
  - getFreeformContentHandlerName
  - getGroupingBlockName
  - getPhrasingContentSchema
  - getPossibleBlockTransformations
  - getSaveContent
  - getSaveElement

- getUnregisteredTypeHandlerName
- hasBlockSupport
- hasChildBlocks
- hasChildBlocksWithInserterSupport
- isReusableBlock
- isTemplatePart
- isUnmodifiedBlock
- isUnmodifiedDefaultBlock
- isValidBlockContent
- isValidIcon
- normalizeIconObject
- parse
- parseWithAttributeSchema
- pasteHandler
- privateApis
- rawHandler
- registerBlockBindingsSource
- registerBlockCollection
- registerBlockStyle
- registerBlockType
- registerBlockVariation
- sanitizeBlockAttributes
- serialize
- serializeRawBlock
- setCategories
- setDefaultBlockName
- setFreeformContentHandlerName
- setGroupingBlockName
- setUnregisteredTypeHandlerName
- store
- switchToBlockType
- synchronizeBlocksWithTemplate
- unregisterBlockBindingsSource
- unregisterBlockStyle

- unregisterBlockType
- unregisterBlockVariation
- updateCategory
- validateBlock
- withBlockContentContext

- Contributing to this package

“Block” is the abstract term used to describe units of markup that, composed together, form the content or layout of a webpage. The idea combines concepts of what in WordPress today we achieve with shortcodes, custom HTML, and embed discovery into a single consistent API and user experience.

For more context, refer to *What Are Little Blocks Made Of?* from the Make WordPress Design blog.

> Learn how to create your first block for the WordPress block editor. From setting up your development environment, tools, and getting comfortable with the new development model, this tutorial covers all you need to know to get started with creating blocks.

Installation

Install the module

```
npm install @wordpress/blocks --save
```

*This package assumes that your code will run in an **ES2015+** environment. If you’re using an environment that has limited or no support for such language features and APIs, you should include the polyfill shipped in `@wordpress/babel-preset-default` in your code.*

API

cloneBlock

Given a block object, returns a copy of the block object, optionally merging new attributes and/or replacing its inner blocks.

Parameters

- *block* `Block`: Block instance.
- *mergeAttributes* `Record< string, unknown >`: Block attributes.
- *newInnerBlocks* `Block[]`: Nested blocks.

Returns

- `Block`: A cloned block.

cloneSanitizedBlock

Given a block object, returns a copy of the block object while sanitizing its attributes, optionally merging new attributes and/or replacing its inner blocks.

Parameters

- *block* `Block`: Block instance.
- *mergeAttributes* `Record< string, unknown >`: Block attributes.
- *newInnerBlocks* `Block[]`: Nested blocks.

Returns

- `Block`: A cloned block.

createBlock

Returns a block object given its type and attributes.

Parameters

- *name* `string`: Block name.
- *attributes* `Record< string, unknown >`: Block attributes.
- *innerBlocks* `Block[]`: Nested blocks.
- *innerContent* `Array< string | null >`: Static HTML fragments interleaved with inner blocks, where `null` entries mark inner block positions. Only applies to the Custom HTML block.

Returns

- `Block`: Block object.

createBlocksFromInnerBlocksTemplate

Given an array of InnerBlocks templates or Block Objects, returns an array of created Blocks from them. It handles the case of having InnerBlocks as Blocks by converting them to the proper format to continue recursively.

Parameters

- *innerBlocksOrTemplate* `Array< Block | TemplateBlock >`: Nested blocks or InnerBlocks templates.

Returns

- `Block[]`: Array of Block objects.

doBlocksMatchTemplate

Checks whether a list of blocks matches a template by comparing the block names.

Parameters

- *blocks* `Block[]`: Block list.
- *template* `TemplateItem[]`: Block template.

Returns

- `boolean`: Whether the list of blocks matches a templates.

findTransform

Given an array of transforms, returns the highest-priority transform where the predicate function returns a truthy value. A higher-priority transform is one with a lower priority value (i.e. first in priority order). Returns null if the transforms set is empty or the predicate function returns a falsey value for all entries.

Parameters

- *transforms* `T[]`: Transforms to search.
- *predicate* `( transform: T ) => boolean`: Function returning true on matching transform.

Returns

- `T | null`: Highest-priority transform candidate.

getBlockAttributes

Returns the block attributes of a registered block node given its type.

Parameters

- *blockTypeOrName* `string | BlockType`: Block type or name.
- *innerHTML* `string | Node | undefined`: Raw block content.
- *attributes* `Record< string, unknown >`: Known block attributes (from delimiters).

Returns

- `Record< string, unknown >`: All block attributes.

getBlockAttributesNamesByRole

Filter block attributes by `role` and return their names.

Parameters

- *name* `string`: Block attribute’s name.
- *role* `string`: The role of a block attribute.

Returns

- `string[]`: The attribute names that have the provided role.

getBlockBindingsSource

Returns a registered block bindings source by its name.

Parameters

- *name* `string`: Block bindings source name.

Returns

- `BlockBindingsSource | undefined`: Block bindings source.

Changelog

`6.7.0` Introduced in WordPress core.

getBlockBindingsSources

Returns all registered block bindings sources.

Returns

- `Record< string, BlockBindingsSource >`: Block bindings sources.

Changelog

`6.7.0` Introduced in WordPress core.

getBlockContent

Given a block object, returns the Block’s Inner HTML markup.

Parameters

- *block* `Block`: Block instance.

Returns

- `string`: HTML.

getBlockDefaultClassName

Returns the block’s default classname from its name.

Parameters

- *blockName* `string`: The block name.

Returns

- `string`: The block’s default class.

getBlockFromExample

Undocumented declaration.

getBlockMenuDefaultClassName

Returns the block’s default menu item classname from its name.

Parameters

- *blockName* `string`: The block name.

Returns

- `string`: The block’s default menu item class.

getBlockSupport

Returns the block support value for a feature, if defined.

Parameters

- *nameOrType* `string | BlockType`: Block name or type object
- *feature* `string | string[]`: Feature to retrieve, as a dot-separated path or an array of path segments
- *defaultSupports* `unknown`: Default value to return if not explicitly defined

Returns

- `unknown`: Block support value

getBlockTransforms

Returns normal block transforms for a given transform direction, optionally for a specific block by name, or an empty array if there are no transforms. If no block name is provided, returns transforms for all blocks. A normal transform object includes `blockName` as a property.

Parameters

- *direction* `'to' | 'from'`: Transform direction (“to”, “from”).
- *blockTypeOrName* `string | BlockType`: Block type or name.

Returns

- `NormalizedBlockTransform[]`: Block transforms for direction.

getBlockType

Returns a registered block type.

Parameters

- *name* `string`: Block name.

Returns

- `BlockType | undefined`: Block type.

getBlockTypes

Returns all registered blocks.

Returns

- `BlockType[]`: Block settings.

getChildBlockNames

Returns an array with the child blocks of a given block.

Parameters

- *blockName* `string`: Name of block (example: “latest-posts”).

Returns

- `string[]`: Array of child block names.

getDefaultBlockName

Retrieves the default block name.

Returns

- `string | null`: Block name.

getFreeformContentHandlerName

Retrieves name of block handling non-block content, or undefined if no handler has been defined.

Returns

- `string | null`: Block name.

getGroupingBlockName

Retrieves name of block used for handling grouping interactions.

Returns

- `string | null`: Block name.

getPhrasingContentSchema

Undocumented declaration.

getPossibleBlockTransformations

Returns an array of block types that the set of blocks received as argument can be transformed into.

Parameters

- *blocks* `Block[]`: Blocks array.

Returns

- `BlockTypeWithTransformMetadata[]`: Block types that the blocks argument can be transformed to.

getSaveContent

Given a block type containing a save render implementation and attributes, returns the static markup to be saved.

Parameters

- *blockTypeOrName* `string | BlockType | undefined | null`: Block type or name.
- *attributes* `Record< string, unknown >`: Block attributes.
- *innerBlocks* `Block[]`: Nested blocks.

Returns

- `string`: Save content.

getSaveElement

Given a block type containing a save render implementation and attributes, returns the enhanced element to be saved or string when raw HTML expected.

Parameters

- *blockTypeOrName* `string | BlockType`: Block type or name.
- *attributes* `Record< string, unknown >`: Block attributes.
- *innerBlocks* `Block[] | ReactElement`: Nested blocks, or an element to render in their place.

Returns

- `unknown`: Save element or raw HTML string.

getUnregisteredTypeHandlerName

Retrieves name of block handling unregistered block types, or undefined if no handler has been defined.

Returns

- `string | null`: Block name.

hasBlockSupport

Returns true if the block defines support for a feature, or false otherwise.

Parameters

- *nameOrType* `string | BlockType`: Block name or type object.
- *feature* `string | string[]`: Feature to test, as a dot-separated path or an array of path segments.
- *defaultSupports* `boolean`: Whether feature is supported by default if not explicitly defined.

Returns

- `boolean`: Whether block supports feature.

hasChildBlocks

Returns a boolean indicating if a block has child blocks or not.

Parameters

- *blockName* `string`: Name of block (example: “latest-posts”).

Returns

- `boolean`: True if a block contains child blocks and false otherwise.

hasChildBlocksWithInserterSupport

Returns a boolean indicating if a block has at least one child block with inserter support.

Parameters

- *blockName* `string`: Block type name.

Returns

- `boolean`: True if a block contains at least one child blocks with inserter support and false otherwise.

isReusableBlock

Determines whether or not the given block is a reusable block. This is a special block type that is used to point to a global block stored via the API.

Parameters

- *blockOrType* `{ name: string; } | null | undefined`: Block or Block Type to test.

Returns

- `boolean`: Whether the given block is a reusable block.

isTemplatePart

Determines whether or not the given block is a template part. This is a special block type that allows composing a page template out of reusable design elements.

Parameters

- *blockOrType* `{ name: string; } | null | undefined`: Block or Block Type to test.

Returns

- `boolean`: Whether the given block is a template part.

isUnmodifiedBlock

Determines whether the block’s attributes are equal to the default attributes which means the block is unmodified.

Parameters

- *block* `Block`: Block Object.
- *role* `string`: Optional role to filter attributes for modification check.

Returns

- `boolean`: Whether the block is an unmodified block.

isUnmodifiedDefaultBlock

Determines whether the block is a default block and its attributes are equal to the default attributes which means the block is unmodified.

Parameters

- *block* `Block`: Block Object
- *role* `string`: Optional role to filter attributes for modification check.

Returns

- `boolean`: Whether the block is an unmodified default block.

isValidBlockContent

> **Deprecated** Use validateBlock instead to avoid data loss.

Returns true if the parsed block is valid given the input content. A block is considered valid if, when serialized with assumed attributes, the content matches the original value.

Logs to console in development environments when invalid.

Parameters

- *blockTypeOrName* `BlockType | string`: Block type.
- *attributes* `Record< string, unknown >`: Parsed block attributes.
- *originalBlockContent* `string`: Original block content.

Returns

- `boolean`: Whether block is valid.

isValidIcon

Function that checks if the parameter is a valid icon.

Parameters

- *icon* `unknown`: Parameter to be checked.

Returns

- `boolean`: True if the parameter is a valid icon and false otherwise.

normalizeIconObject

Function that receives an icon as set by the blocks during the registration and returns a new icon object that is normalized so we can rely on just on possible icon structure in the codebase.

Parameters

- *icon* `BlockTypeIcon | undefined`: Render behavior of a block type icon; one of a Dashicon slug, an element, or a component.

Returns

- `BlockTypeIconDescriptor`: Object describing the icon.

parse

Utilizes an optimized token-driven parser based on the Gutenberg grammar spec defined through a parsing expression grammar to take advantage of the regular cadence provided by block delimiters — composed syntactically through HTML comments — which, given a general HTML document as an input, returns a block list array representation.

This is a recursive-descent parser that scans linearly once through the input document. Instead of directly recursing it utilizes a trampoline mechanism to prevent stack overflow. This initial pass is

mainly interested in separating and isolating the blocks serialized in the document and manifestly not in the content within the blocks.

Related

- https://developer.wordpress.org/block-editor/packages/packages-block-serialization-default-parser/

Parameters

- *content* `string`: The post content.
- *options* `ParseOptions`: Extra options for handling block parsing.

Returns

- `Block[]`: Block list.

parseWithAttributeSchema

Given a block's raw content and an attribute's schema returns the attribute's value depending on its source.

Parameters

- *innerHTML* `string | Node | undefined`: Block's raw content.
- *attributeSchema* `BlockAttribute`: Attribute's schema.

Returns

- `unknown`: Attribute value.

pasteHandler

Converts an HTML string to known blocks. Strips everything else.

Parameters

- *options* `RawHandlerOptions`:
- *options.HTML* `RawHandlerOptions[ 'HTML' ]`: The HTML to convert.
- *options.plainText* `RawHandlerOptions[ 'plainText' ]`: Plain text version.
- *options.mode* `RawHandlerOptions[ 'mode' ]`: Handle content as blocks or inline content. _ 'AUTO': Decide based on the content passed. _ 'INLINE': Always handle as inline content, and return string. * 'BLOCKS': Always handle as blocks, and return array of blocks.

- *options.tagName* `RawHandlerOptions[ 'tagName' ]`: The tag into which content will be inserted.

Returns

- `Block[] | string`: A list of blocks or a string, depending on `handlerMode`.

privateApis

Undocumented declaration.

rawHandler

Converts an HTML string to known blocks.

Usage

```
import { rawHandler } from '@wordpress/blocks';


const blocks = rawHandler( { HTML: '<p>Hello</p><p>World</p>' } );
```

Parameters

- *options* `{ HTML?: string; }`: Options.
- *options.HTML* `string`: The HTML to convert.

Returns

- `Block[]`: A list of blocks.

registerBlockBindingsSource

Registers a new block bindings source with an object defining its behavior. Once registered, the source is available to be connected to the supported block attributes.

Usage

```
import { _x } from '@wordpress/i18n';
import { registerBlockBindingsSource } from '@wordpress/blocks';


registerBlockBindingsSource( {
```

```
    name: 'plugin/my-custom-source',
    label: _x( 'My Custom Source', 'block bindings source' ),
    usesContext: [ 'postType' ],
    getValues: getSourceValues,
    setValues: updateMyCustomValuesInBatch,
    canUserEditValue: () => true,
} );
```

Parameters

- *source* `BlockBindingsSource`: Object describing a block bindings source.

Changelog

`6.7.0` Introduced in WordPress core.

registerBlockCollection

Registers a new block collection to group blocks in the same namespace in the inserter.

Usage

```
import { __ } from '@wordpress/i18n';
import { registerBlockCollection, registerBlockType } from '@wordpress/blocks';

// Register the collection.
registerBlockCollection( 'my-collection', {
    title: __( 'Custom Collection' ),
} );

// Register a block in the same namespace to add it to the collection.
registerBlockType( 'my-collection/block-name', {
    title: __( 'My First Block' ),
    edit: () => <div>{ __( 'Hello from the editor!' ) }</div>,
    save: () => <div>'Hello from the saved content!</div>,
} );
```

Parameters

- *namespace* `string`: The namespace to group blocks by in the inserter; corresponds to the block namespace.
- *settings* `{ title: string; icon?: Icon; }`: The block collection settings.

- *settings.title* `string`: The title to display in the block inserter.
- *settings.icon* `Icon`: The icon to display in the block inserter.

registerBlockStyle

Registers a new block style for the given block types.

For more information on connecting the styles with CSS the official documentation.

Usage

```
import { __ } from '@wordpress/i18n';
import { registerBlockStyle } from '@wordpress/blocks';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    return (
        <Button
            onClick={ () => {
                registerBlockStyle( 'core/quote', {
                    name: 'fancy-quote',
                    label: __( 'Fancy Quote' ),
                } );
            } }
        >
            { __( 'Add a new block style for core/quote' ) }
        </Button>
    );
};
```

Parameters

- *blockNames* `string | string[]`: Name of blocks e.g. “core/latest-posts” or `[“core/group”, “core/columns”]`.
- *styleVariation* `BlockStyle | BlockStyle[]`: Object containing `name` which is the class name applied to the block and `label` which identifies the variation to the user.

registerBlockType

Registers a new block provided a unique name and an object defining its behavior. Once registered, the block is made available as an option to any editor interface where blocks are

implemented.

For more in-depth information on registering a custom block see the Create a block tutorial.

Usage

```
import { __ } from '@wordpress/i18n';
import { registerBlockType } from '@wordpress/blocks';

registerBlockType( 'namespace/block-name', {
    title: __( 'My First Block' ),
    edit: () => <div>{ __( 'Hello from the editor!' ) }</div>,
    save: () => <div>Hello from the saved content!</div>,
} );
```

Parameters

- *blockNameOrMetadata* `string | BlockConfiguration< Attributes >`: Block type name or its metadata.
- *settings* `Partial< SettingsBlockConfiguration< Attributes > >`: Block settings.

Returns

- `BlockType | undefined`: The block, if it has been successfully registered; otherwise `undefined`.

registerBlockVariation

Registers a new block variation for the given block type.

For more information on block variations see the official documentation .

Usage

```
import { __ } from '@wordpress/i18n';
import { registerBlockVariation } from '@wordpress/blocks';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    return (
        <Button
            onClick={ () => {
```

```
                registerBlockVariation( 'core/embed', {
                    name: 'custom',
                    title: __( 'My Custom Embed' ),
                    attributes: { providerNameSlug: 'custom' },
                } );
            } }
        >
            __( 'Add a custom variation for core/embed' ) }
        </Button>
    );
};
```

Parameters

- *blockName* `string`: Name of the block (example: “core/columns”).
- *variation* `BlockVariation | BlockVariation[]`: Object describing a block variation.

sanitizeBlockAttributes

Ensure attributes contains only values defined by block type, and merge default values for missing attributes.

Parameters

- *name* `string`: The block’s name.
- *attributes* `Record< string, unknown >`: The block’s attributes.

Returns

- `Record< string, unknown >`: The sanitized attributes.

serialize

Takes a block or set of blocks and returns the serialized post content.

Parameters

- *blocks* `Block | Block[]`: Block(s) to serialize.
- *options* `BlockSerializationOptions`: Serialization options.

Returns

- `string`: The post content.

serializeRawBlock

Serializes a block node into the native HTML-comment-powered block format. CAVEAT: This function is intended for re-serializing blocks as parsed by valid parsers and skips any validation steps. This is NOT a generic serialization function for in-memory blocks. For most purposes, see the following functions available in the `@wordpress/blocks` package:

Related

- serializeBlock
- serialize For more on the format of block nodes as returned by valid parsers:
- `@wordpress/block-serialization-default-parser` package
- `@wordpress/block-serialization-spec-parser` package

Parameters

- *rawBlock* `RawBlock`: A block node as returned by a valid parser.
- *options* `[SerializeRawBlockOptions]`: Serialization options.

Returns

- `string`: An HTML string representing a block.

setCategories

Sets the block categories.

Usage

```
import { __ } from '@wordpress/i18n';
import { store as blocksStore, setCategories } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    // Retrieve the list of current categories.
    const blockCategories = useSelect(
        ( select ) => select( blocksStore ).getCategories(),
        []
    );
```

```
    return (
        <Button
            onClick={ () => {
                // Add a custom category to the existing list.
                setCategories( [
                    ...blockCategories,
                    { title: 'Custom Category', slug: 'custom-category' },
                ] );
            } }
        >
            { __( 'Add a new custom block category' ) }
        </Button>
    );
};
```

Parameters

- *categories* `BlockCategory[]`: Block categories.

setDefaultBlockName

Assigns the default block name.

Usage

```
import { setDefaultBlockName } from '@wordpress/blocks';

const ExampleComponent = () => {
    return (
        <Button onClick={ () => setDefaultBlockName( 'core/heading' ) }>
            { __( 'Set the default block to Heading' ) }
        </Button>
    );
};
```

Parameters

- *name* `string`: Block name.

setFreeformContentHandlerName

Assigns name of block for handling non-block content.

Parameters

- *blockName* `string`: Block name.

setGroupingBlockName

Assigns name of block for handling block grouping interactions.

This function lets you select a different block to group other blocks in instead of the default `core/group` block. This function must be used in a component or when the DOM is fully loaded. See https://developer.wordpress.org/block-editor/reference-guides/packages/packages-dom-ready/

Usage

```
import { setGroupingBlockName } from '@wordpress/blocks';

const ExampleComponent = () => {
    return (
        <Button onClick={ () => setGroupingBlockName( 'core/columns' ) }>
            { __( 'Wrap in columns' ) }
        </Button>
    );
};
```

Parameters

- *name* `string`: Block name.

setUnregisteredTypeHandlerName

Assigns name of block handling unregistered block types.

Parameters

- *blockName* `string`: Block name.

store

Store definition for the blocks namespace.

Related

- https://github.com/WordPress/gutenberg/blob/HEAD/packages/data/README.md#createReduxStore

switchToBlockType

Switch one or more blocks into one or more blocks of the new block type.

Parameters

- *blocks* `Block[] | Block`: Blocks array or block object.
- *name* `string`: Block name.
- *variationName* `string`: Optional target block variation name.

Returns

- `Block[] | null`: Array of blocks or null.

synchronizeBlocksWithTemplate

Synchronize a block list with a block template.

Synchronizing a block list with a block template means that we loop over the blocks keep the block as is if it matches the block at the same position in the template (If it has the same name) and if doesn't match, we create a new block based on the template. Extra blocks not present in the template are removed.

Parameters

- *blocks* `Block[]`: Block list.
- *template* `TemplateItem[]`: Block template.

Returns

- `Block[]`: Updated Block list.

unregisterBlockBindingsSource

Unregisters a block bindings source by providing its name.

Usage

```
import { unregisterBlockBindingsSource } from '@wordpress/blocks';


unregisterBlockBindingsSource( 'plugin/my-custom-source' );
```

Parameters

- *name* `string`: The name of the block bindings source to unregister.

Changelog

`6.7.0` Introduced in WordPress core.

unregisterBlockStyle

Unregisters a block style for the given block.

Usage

```
import { __ } from '@wordpress/i18n';
import { unregisterBlockStyle } from '@wordpress/blocks';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    return (
        <Button
            onClick={ () => {
                unregisterBlockStyle( 'core/quote', 'plain' );
            } }
        >
            { __( 'Remove the "Plain" block style for core/quote' ) }
        </Button>
    );
};
```

Parameters

- *blockName* `string`: Name of block (example: “core/latest-posts”).
- *styleVariationName* `string`: Name of class applied to the block.

unregisterBlockType

Unregisters a block.

Usage

```
import { __ } from '@wordpress/i18n';
import { unregisterBlockType } from '@wordpress/blocks';

const ExampleComponent = () => {
    return (
        <Button
            onClick={ () => unregisterBlockType( 'my-collection/block-name' ) }
        >
            { __( 'Unregister my custom block.' ) }
        </Button>
    );
};
```

Parameters

- *name* `string`: Block name.

Returns

- `BlockType | undefined`: The previous block value, if it has been successfully unregistered; otherwise `undefined`.

unregisterBlockVariation

Unregisters a block variation defined for the given block type.

Usage

```
import { __ } from '@wordpress/i18n';
import { unregisterBlockVariation } from '@wordpress/blocks';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    return (
        <Button
```

```
            onClick={ () => {
                unregisterBlockVariation( 'core/embed', 'youtube' );
            } }
        >
            { __( 'Remove the YouTube variation from core/embed' ) }
        </Button>
    );
};
```

Parameters

- *blockName* `string`: Name of the block (example: “core/columns”).
- *variationName* `string | string[]`: Name of the variation defined for the block.

updateCategory

Updates a category.

Usage

```
import { __ } from '@wordpress/i18n';
import { updateCategory } from '@wordpress/blocks';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    return (
        <Button
            onClick={ () => {
                updateCategory( 'text', { title: __( 'Written Word' ) } );
            } }
        >
            { __( 'Update Text category title' ) }
        </Button>
    );
};
```

Parameters

- *slug* `string`: Block category slug.
- *category* `Partial< BlockCategory >`: Object containing the category properties that should be updated.

validateBlock

Returns an object with `isValid` property set to `true` if the parsed block is valid given the input content. A block is considered valid if, when serialized with assumed attributes, the content matches the original value. If block is invalid, this function returns all validations issues as well.

Parameters

- *block* `Block`: Block object.
- *blockTypeOrName* `BlockType | string`: Block type or name, inferred from block if not given.

Returns

- `[ boolean, LoggerItem[] ]`: Validation results.

withBlockContentContext

> **Deprecated**

A Higher Order Component used to inject BlockContent using context to the wrapped component.

Parameters

- *OriginalComponent* `T`: The component to enhance.

Returns

- `T`: The same component.

Contributing to this package

This is an individual package that's part of the Gutenberg project. The project is organized as a monorepo. It's made up of multiple self-contained software packages, each with a specific purpose. The packages in this monorepo are published to npm and used by WordPress as well as other software projects.

To find out more about contributing to this package or Gutenberg as a whole, please read the project's main contributor guide.

□□□ #2
□ □□□ □□□ 2026-10-31 18:17:50 UTC
□ □□□ □□□ 2026-10-31 18:17:53 UTC