# Attributes


- Type validation
- Enum validation
- Value source
  - attribute source
  - text source
  - html source
  - query source
  - Meta source (deprecated)
    - Considerations
- Default value
- Role


Block attributes provide information about the data stored by a block. For example, rich content, a list of image URLs, a background color, or a button title.

A block can contain any number of attributes, and these are specified by the `attributes` field – an object where each key is the name of the attribute, and the value is the attribute definition.

The attribute definition will contain, at a minimum, either a `type` or an `enum`. There may be additional fields.

*Example*: Attributes object defining three attributes – `url`, `title`, and `size`.

```
{
    url: {
        type: 'string',
        source: 'attribute',
        selector: 'img',
        attribute: 'src',
    },
    title: {
        type: 'string',
```

```
    },
    size: {
        enum: [ 'large', 'small' ],
    },
}
```

When a block is parsed this definition will be used to extract data from the block content. Anything that matches will be available to your block through the `attributes` prop.

This parsing process can be summarized as:

1. Extract value from the `source`.
2. Check value matches the `type`, or is one of the `enum` values.

*Example*: Attributes available in the `edit` and function, using the above attributes definition.

```
function YourBlockEdit( { attributes } ) {
    return (
        <p>URL is { attributes.url }, title is { attributes.title }, and size is {
attributes.size }.</p>
    )
}
```

The block is responsible for using the `save` function to ensure that all attributes with a `source` field are saved according to the attributes definition. This is not automatic.

Attributes without a `source` will be automatically saved in the block comment delimiter.

For example, using the above attributes definition you would need to ensure that your `save` function has a corresponding img tag for the `url` attribute. The `title` and `size` attributes will be saved in the comment delimiter.

*Example*: Example `save` function that contains the `url` attribute.

```
function YourBlockSave( { attributes } ) {
    return (
        <img src={ attributes.url } />
    )
}
```

The saved HTML will contain the `title` and `size` in the comment delimiter, and the `url` in the `img` node.

```
<!-- block:your-block {"title":"hello world","size":"large"} -->
<img src="/image.jpg" />
<!-- /block:your-block -->
```

If an attribute changes over time then a block deprecation can help migrate from an older attribute, or remove it entirely.

# Type validation

The `type` indicates the type of data that is stored by the attribute. It does not indicate where the data is stored, which is defined by the `source` field.

A `type` is required, unless an `enum` is provided. A `type` can be used with an `enum`.

The `type` field MUST be one of the following:

- `null`
- `boolean`
- `object`
- `array`
- `string`
- `integer`
- `number` (same as `integer`)

Note that the validity of an `object` is determined by your `source`. For an example, see the `query` details below.

# Enum validation

An attribute can be defined as one of a fixed set of values. This is specified by an `enum`, which contains an array of allowed values:

*Example*: Example `enum`.

```
{
    size: {
        enum: [ 'large', 'small', 'tiny' ]
    }
}
```

# Value source

Attribute sources are used to define how the attribute values are extracted from saved post content. They provide a mechanism to map from the saved markup to a JavaScript representation of a block.

The available `source` values are:

– `(no value)` – when no `source` is specified then data is stored in the block's comment delimiter.
– `attribute` – data is stored in an HTML element attribute.
– `text` – data is stored in HTML text.
– `html` – data is stored in HTML. This is typically used by `RichText`.
– `query` – data is stored as an array of objects.
– `meta` – data is stored in post meta (deprecated).

The `source` field is usually combined with a `selector` field. If no selector argument is specified, the source definition runs against the block's root node. If a selector argument is specified, it will run against the matching element(s) within the block.

The `selector` can be an HTML tag, or anything queryable with querySelector, such as a class or id attribute. Examples are given below.

For example, a `selector` of `img` will match an `img` element, and `img.class` will match an `img` element that has a class of `class`.

Under the hood, attribute sources are a superset of the functionality provided by hpq, a small library used to parse and query HTML markup into an object shape.

To summarize, the `source` determines where data is stored in your content, and the `type` determines what that data is. To reduce the amount of data stored it is usually better to store as much data as possible within HTML rather than as attributes within the comment delimiter.

## `attribute` source

Use an `attribute` source to extract the value from an attribute in the markup. The attribute is specified by the `attribute` field, which must be supplied.

*Example*: Extract the `src` attribute from an image found in the block's markup.

Saved content:

```
<div>
    Block Content
```

```
    <img src="https://lorempixel.com/1200/800/" />
</div>
```

Attribute definition:

```
{
    url: {
        type: 'string',
        source: 'attribute',
        selector: 'img',
        attribute: 'src',
    }
}
```

Attribute available in the block:

```
{ "url": "https://lorempixel.com/1200/800/" }
```

Most attributes from markup will be of type `string`. Numeric attributes in HTML are still stored as strings, and are not converted automatically.

*Example*: Extract the `width` attribute from an image found in the block's markup.

Saved content:

```
<div>
    Block Content

    <img src="https://lorempixel.com/1200/800/" width="50" />
</div>
```

Attribute definition:

```
{
    width: {
        type: 'string',
        source: 'attribute',
        selector: 'img',
        attribute: 'width',
    }
}
```

Attribute available in the block:

```
{ "width": "50" }
```

The only exception is when checking for the existence of an attribute (for example, the `disabled` attribute on a `button`). In that case type `boolean` can be used and the stored value will be a boolean.

*Example*: Extract the `disabled` attribute from a button found in the block's markup.

Saved content:

```
<div>
    Block Content

    <button type="button" disabled>Button</button>
</div>
```

Attribute definition:

```
{
    disabled: {
        type: 'boolean',
        source: 'attribute',
        selector: 'button',
        attribute: 'disabled',
    }
}
```

Attribute available in the block:

```
{ "disabled": true }
```

## `text` source

Use `text` to extract the inner text from markup. Note that HTML is returned according to the rules of `textContent`.

*Example*: Extract the `content` attribute from a figcaption element found in the block's markup.

Saved content:

```
<figure>
    <img src="/image.jpg" />

    <figcaption>The inner text of the figcaption element</figcaption>
</figure>
```

Attribute definition:

```
{
    content: {
        type: 'string',
        source: 'text',
        selector: 'figcaption',
    }
}
```

Attribute available in the block:

```
{ "content": "The inner text of the figcaption element" }
```

Another example, using `text` as the source, and using `.my-content` class as the selector to extract text:

*Example*: Extract the `content` attribute from an element with `.my-content` class found in the block's markup.

Saved content:

```
<div>
    <img src="/image.jpg" />

    <p class="my-content">The inner text of .my-content class</p>
</div>
```

Attribute definition:

```
{
    content: {
        type: 'string',
        source: 'text',
        selector: '.my-content',
    }
```

```
}
```

Attribute available in the block:

```
{ "content": "The inner text of .my-content class" }
```

## `html` source

Use `html` to extract the inner HTML from markup. Note that text is returned according to the rules of `innerHTML`.

*Example*: Extract the `content` attribute from a figcaption element found in the block's markup.

Saved content:

```
<figure>
    <img src="/image.jpg" />

    <figcaption>The inner text of the <strong>figcaption</strong> element</figcaption>
</figure>
```

Attribute definition:

```
{
    content: {
        type: 'string',
        source: 'html',
        selector: 'figcaption',
    }
}
```

Attribute available in the block:

```
{ "content": "The inner text of the <strong>figcaption</strong> element" }
```

## `query` source

Use `query` to extract an array of values from markup. Entries of the array are determined by the `selector` argument, where each matched element within the block will have an entry structured corresponding to the second argument, an object of attribute sources.

The `query` field is effectively a nested block attributes definition. It is possible (although not necessarily recommended) to nest further.

*Example*: Extract `src` and `alt` from each image element in the block's markup.

Saved content:

```
<div>
    <img src="https://lorempixel.com/1200/800/" alt="large image" />
    <img src="https://lorempixel.com/50/50/" alt="small image" />
</div>
```

Attribute definition:

```
{
    images: {
        type: 'array',
        source: 'query',
        selector: 'img',
        query: {
            url: {
                type: 'string',
                source: 'attribute',
                attribute: 'src',
            },
            alt: {
                type: 'string',
                source: 'attribute',
                attribute: 'alt',
            },
        }
    }
}
```

Attribute available in the block:

```
{
    "images": [
        { "url": "https://lorempixel.com/1200/800/", "alt": "large image" },
        { "url": "https://lorempixel.com/50/50/", "alt": "small image" }
    ]
```

```
}
```

# Meta source (deprecated)

> Although attributes may be obtained from a post's meta, meta attribute sources are considered deprecated; EntityProvider and related hook APIs should be used instead, as shown in the Create Meta Block how-to.

Attributes may be obtained from a post's meta rather than from the block's representation in saved post content. For this, an attribute is required to specify its corresponding meta key under the `meta` key.

Attribute definition:

```
{
    author: {
        type: 'string',
        source: 'meta',
        meta: 'author'
    },
},
```

From here, meta attributes can be read and written by a block using the same interface as any attribute:

```
edit( { attributes, setAttributes } ) {
    function onChange( event ) {
        setAttributes( { author: event.target.value } );
    }

    return <input value={ attributes.author } onChange={ onChange } type="text" />;
},
```

## Considerations

By default, a meta field will be excluded from a post object's meta. This can be circumvented by explicitly making the field visible:

```
function gutenberg_my_block_init() {
    register_post_meta( 'post', 'author', array(
        'show_in_rest' => true,
    ) );
}
add_action( 'init', 'gutenberg_my_block_init' );
```

Furthermore, be aware that WordPress defaults to:

- not treating a meta datum as being unique, instead returning an array of values;
- treating that datum as a string.

If either behavior is not desired, the same `register_post_meta` call can be complemented with the `single` and/or `type` parameters as follows:

```
function gutenberg_my_block_init() {
    register_post_meta( 'post', 'author_count', array(
        'show_in_rest' => true,
        'single' => true,
        'type' => 'integer',
    ) );
}
add_action( 'init', 'gutenberg_my_block_init' );
```

If you'd like to use an object or an array in an attribute, you can register a `string` attribute type and use JSON as the intermediary. Serialize the structured data to JSON prior to saving, and then deserialize the JSON string on the server. Keep in mind that you're responsible for the integrity of the data; make sure to properly sanitize, accommodate missing data, etc.

Lastly, make sure that you respect the data's type when setting attributes, as the framework does not automatically perform type casting of meta. Incorrect typing in block attributes will result in a post remaining dirty even after saving (*cf.* `isEditedPostDirty`, `hasEditedAttributes`). For instance, if `authorCount` is an integer, remember that event handlers may pass a different kind of data, thus the value should be cast explicitly:

```
function onChange( event ) {
    props.setAttributes( { authorCount: Number( event.target.value ) } );
}
```

# Default value

A block attribute can contain a default value, which will be used if the `type` and `source` do not match anything within the block content.

The value is provided by the `default` field, and the value should match the expected format of the attribute.

*Example*: Example `default` values.

```
{
    type: 'string',
    default: 'hello world'
}
```

```
{
    type: 'array',
    default: [
        { "url": "https://lorempixel.com/1200/800/", "alt": "large image" },
        { "url": "https://lorempixel.com/50/50/", "alt": "small image" }
    ]
}
```

```
{
    type: 'object',
    default: {
        width: 100,
        title: 'title'
    }
}
```

# Role

The `role` property designates an attribute as being of a particular conceptual type. This property can be applied to any attribute to provide semantic meaning about how the attribute should be handled.

Use `content` to designate the attribute as user-editable content. Blocks with attributes marked as `content` may be enabled for privileged editing in special circumstances such as content only locking.
Use `local` to mark the attribute as temporary and non-persistable. Attributes marked as `local` are ignored by the Block Serializer and never saved to post content.

*Example*: `content` role used by the paragraph block.

```
{
    content: {
        type: 'string',
        source: 'html',
        selector: 'p',
        role: 'content',
    }
}
```

*Example*: `local` role used for temporary data.

```
{
    blob: {
        type: 'string',
        role: 'local',
    }
}
```

Learn more in the WordPress 6.7 dev note.

---

□□□ #2
□ □□□ □□□ 2026-10-31 18:10:47 UTC
□ □□□ □□□ 2026-10-31 18:10:49 UTC