Coding Guidelines

- CSS
  - Naming
    - SCSS file naming conventions for blocks
- JavaScript
  - Imports
    - External dependencies
    - WordPress dependencies
    - Internal dependencies
  - Legacy experimental APIs, plugin-only APIs, and private APIs
    - Legacy experimental APIs
    - Plugin-only APIs
    - Private APIs
      - Private selectors and actions
      - Private functions, classes, and variables
    - Private function arguments
    - Private React component properties
    - Private editor settings
    - Private block.json and theme.json APIs
    - Inline small actions in thunks
  - Exposing private APIs publicly
  - Objects
  - Strings
  - Optional chaining
  - React components
- JavaScript documentation using JSDoc
  - Custom types
  - Importing and exporting types
    - External dependencies

    - Generic types
    - Nullable, undefined, and void types
    - Documenting examples
    - Documenting React components
- PHP
- GitHub Actions workflow files

This living document serves to prescribe coding guidelines specific to the Gutenberg project. Base coding guidelines follow the WordPress Coding Standards. The following sections outline additional patterns and conventions used in the Gutenberg project.

CSS

Naming

To avoid class name collisions, class names **must** adhere to the following guidelines, which are loosely inspired by the BEM (Block, Element, Modifier) methodology.

All class names assigned to an element must be prefixed with the name of the package, followed by a dash and the name of the directory in which the component resides. Any descendent of the component's root element must append a dash-delimited descriptor, separated from the base by two consecutive underscores `__`.

- Root element: `package-directory`
- Child elements: `package-directory__descriptor-foo-bar`

The root element is considered to be the highest ancestor element returned by the default export in the `index.js`. Notably, if your folder contains multiple files, each with their own default exported component, only the element rendered by that of `index.js` can be considered the root. All others should be treated as descendents.

Example:

Consider the following component located at `packages/components/src/notice/index.js`:

```
export default function Notice( { children, onRemove, actions = [] } ) {
    return (
        <div className="components-notice">
```

```
            <div className="components-notice__content">{ children }</div>
            { actions.length > 0 && (
                <div className="components-notice__actions">
                    { /* action buttons */ }
                </div>
            ) }
            <Button
                className="components-notice__dismiss"
                icon={ closeSmall }
                label={ __( 'Dismiss this notice' ) }
                onClick={ onRemove }
            />
        </div>
    );
}
```

Optional regions exposed via props should use additional `__descriptor` classes on direct descendants of the root, following the same pattern as the other child elements in the example above.

Components may be assigned with class names that indicate states (for example, an “active” tab or an “opened” panel). These modifiers should be applied as a separate class name, prefixed as an adjective expression by `is-` (`is-active` or `is-opened`). In rare cases, you may encounter variations of the modifier prefix, usually to improve readability (`has-warning`). Because a modifier class name is not contextualized to a specific component, it should always be written in stylesheets as accompanying the component being modified (`.components-panel.is-opened`).

Example:

Consider a panel that can be expanded. The `clsx` package can be a helpful utility for conditionally applying modifier class names.

```
import clsx from 'clsx';

export default function Panel( { children, isExpanded } ) {
    const classes = clsx( 'components-panel', {
        'is-expanded': isExpanded,
    } );

    return <div className={ classes }>{ /* ... */ }</div>;
}
```

A component's class name should **never** be used outside its own folder (with rare exceptions such as `_z-index.scss`). If you need to inherit styles of another component in your own components, you should render an instance of that other component. At worst, you should duplicate the styles within your own component's stylesheet. This is intended to improve maintainability by isolating shared components as a reusable interface, reducing the surface area of similar UI elements by adapting a limited set of common components to support a varied set of use-cases.

SCSS file naming conventions for blocks

The build process will split SCSS from within the blocks library directory into two separate CSS files when Webpack runs.

Styles placed in a `style.scss` file will be built into `blocks/build/style.css`, to load on the front end theme as well as in the editor. If you need additional styles specific to the block's display in the editor, add them to an `editor.scss`.

Examples of styles that appear in both the theme and the editor include gallery columns and drop caps.

JavaScript

All new code in Gutenberg should be written in TypeScript, with a `.ts` file extension, or a `.tsx` file extension for files including JSX syntax.

There are some exceptions where writing plain JavaScript (`.js`) is permitted:

- If the code is expected to be run directly in the browser or by Node.js in environments where TypeScript syntax is not supported. Note that as of Node.js v22.18.0 and newer, TypeScript files containing erasable syntax can be executed directly by the Node.js runtime.
- If authoring new code in an existing `.js` or `.jsx` file. Migrating existing files to the equivalent `.ts` or `.tsx` is encouraged if it is trivial to do so.
- If authoring new files in a package that has no `tsconfig.json` yet, such that the new file would not have reasonable access to existing package typings. Packages with a `tsconfig.json` reject new JavaScript files (`.js`, `.jsx`, `.mjs`, `.cjs`) under `src` and `test` through the `gutenberg/use-typescript` ESLint rule, as do `routes/` and `tools/`, with the files that predate the rule listed in `tools/eslint/suppressions.json`.

JavaScript in Gutenberg uses modern language features of the ECMAScript language specification as well as the JSX language syntax extension. These are enabled through a combination of preset configurations, notably `@wordpress/babel-preset-default` which is used as a preset in the project's Babel configuration.

While the staged process for introducing a new JavaScript language feature offers an opportunity to use new features before they are considered complete, **the Gutenberg project and the `@wordpress/babel-preset-default` configuration will only target support for proposals which have reached Stage 4 ("Finished")**.

Imports

In the Gutenberg project, we use the ES2015 import syntax to enable us to create modular code with clear separations between code of a specific feature, code shared across distinct WordPress features, and third-party dependencies.

Imports can reference third-party packages, WordPress packages, or local files.

Write imports as one contiguous block. Do not separate dependency types with comments or blank lines.

External dependencies

An external dependency is third-party code that is not maintained by WordPress contributors, but instead included in WordPress as a default script or referenced from an outside package manager like npm.

Example:

```
import moment from 'moment';
```

WordPress dependencies

To encourage reusability between features, our JavaScript is split into domain-specific modules which `export` one or more functions or objects. In the Gutenberg project, we've distinguished these modules under top-level directories. Each module serve an independent purpose, and often code is shared between them. For example, in order to localize its text, editor code will need to include functions from the `i18n` module.

Example:

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

Internal dependencies

Within a specific feature, code is organized into separate files and folders. As is the case with external and WordPress dependencies, you can bring this code into scope by using the `import` keyword. The main distinction here is that when importing internal files, you should use relative

paths specific to top-level directory you're working in.

Example:

```
import VisualEditor from '../visual-editor';
```

Legacy experimental APIs, plugin-only APIs, and private APIs

Legacy experimental APIs

Historically, Gutenberg has used the `__experimental` and `__unstable` prefixes to indicate that a given API is not yet stable and may be subject to change. This is a legacy convention which should be avoided in favor of the plugin-only API pattern or a private API pattern described below.

The problem with using the prefixes was that these APIs rarely got stabilized or removed. As of June 2022, WordPress Core contained 280 publicly exported experimental APIs merged from the Gutenberg plugin during the major WordPress releases. Many plugins and themes started relying on these experimental APIs for essential features that couldn't be accessed in any other way.

The legacy `__experimental` APIs can't be removed on a whim anymore. They became a part of the WordPress public API and fall under the WordPress Backwards Compatibility policy. Removing them involves a deprecation process. It may be relatively easy for some APIs, but it may require effort and span multiple WordPress releases for others.

All in all, don't use the `__experimental` prefix for new APIs. Use plugin-only APIs and private APIs instead.

Plugin-only APIs

Plugin-only APIs are temporary values exported from a module whose existence is either pending future revision or provides an immediate means to an end.

To External Consumers:

There is no support commitment for plugin-only APIs. They can and will be removed or changed without advance warning, including as part of a minor or patch release. As an external consumer, you should avoid these APIs.

To Project Contributors:

An **plugin-only API** is one which is planned for eventual public availability, but is subject to further experimentation, testing, and discussion. It should be made stable or removed at the earliest

opportunity.

Plugin-only APIs are excluded from WordPress Core and only available in the Gutenberg Plugin:

```
// Using globalThis.IS_GUTENBERG_PLUGIN allows Webpack to exclude this
// export from WordPress core:
if ( globalThis.IS_GUTENBERG_PLUGIN ) {
    export { doSomethingExciting } from './api';
}
```

The public interface of such APIs is not yet finalized. Aside from references within the code, these APIs should neither be documented nor mentioned in any CHANGELOG. They should effectively be considered to not exist from an external perspective. In most cases, they should only be exposed to satisfy requirements between packages maintained in this repository.

While a plugin-only API may often stabilize into a publicly-available API, there is no guarantee that it will.

Private APIs

Each `@wordpress` package wanting to privately access or expose private APIs can do so by opting-in to `@wordpress/private-apis`:

```
// In packages/block-editor/private-apis.js:
import { __dangerousOptInToUnstableAPIsOnlyForCoreModules } from '@wordpress/private-apis';
export const { lock, unlock } =
    __dangerousOptInToUnstableAPIsOnlyForCoreModules(
        'I acknowledge private features are not for use in themes or plugins and doing so will
break in the next version of WordPress.',
        '@wordpress/block-editor' // Name of the package calling
__dangerousOptInToUnstableAPIsOnlyForCoreModules,
        // (not the name of the package whose APIs you want to access)
    );
```

Each `@wordpress` package may only opt-in once. The process clearly communicates the extenders are not supposed

to use it. This document will focus on the usage examples, but you can find out more about the `@wordpress/private-apis` package in the its README.md.

Once the package opted-in, you can use the `lock()` and `unlock()` utilities:

```
// Say this object is exported from a package:
export const publicObject = {};

// However, this string is internal and should not be publicly available:
const privateString = 'private information';

// Solution: lock the string "inside" of the object:
lock( publicObject, privateString );

// The string is not nested in the object and cannot be extracted from it:
console.log( publicObject );
// {}

// The only way to access the string is by "unlocking" the object:
console.log( unlock( publicObject ) );
// "private information"

// lock() accepts all data types, not just strings:
export const anotherObject = {};
lock( anotherObject, function privateFn() {} );
console.log( unlock( anotherObject ) );
// function privateFn() {}
```

Keep reading to learn how to use `lock()` and `unlock()` to avoid publicly exporting different kinds of `private` APIs.

Private selectors and actions

You can attach private selectors and actions to a public store:

```
// In packages/package1/store.js:
import { privateHasContentRoleAttribute } from './private-selectors';
import { privateToggleFeature } from './private-actions';
// The `lock` function is exported from the internal private-apis.js file where
// the opt-in function was called.
import { lock, unlock } from './lock-unlock';

export const store = registerStore( /* ... */ );
// Attach a private action to the exported store:
unlock( store ).registerPrivateActions( {
    privateToggleFeature,
```

```
} );

// Attach a private action to the exported store:
unlock( store ).registerPrivateSelectors( {
    privateHasContentRoleAttribute,
} );
```

```
// In packages/package2/MyComponent.js:
import { store } from '@wordpress/package1';
import { useSelect } from '@wordpress/data';
// The `unlock` function is exported from the internal private-apis.js file where
// the opt-in function was called.
import { unlock } from './lock-unlock';

function MyComponent() {
    const hasRole = useSelect(
        ( select ) =>
            // Use the private selector:
            unlock( select( store ) ).privateHasContentRoleAttribute()
        // Note the unlock() is required. This line wouldn't work:
        // select( store ).privateHasContentRoleAttribute()
    );

    // Use the private action:
    unlock( useDispatch( store ) ).privateToggleFeature();

    // ...
}
```

Private functions, classes, and variables

```
// In packages/package1/index.js:
import { lock } from './lock-unlock';

export const privateApis = {};
/* Attach private data to the exported object */
lock( privateApis, {
    privateCallback: function () {},
    privateReactComponent: function PrivateComponent() {
        return <div />;
```

```
    },
    privateClass: class PrivateClass {},
    privateVariable: 5,
} );
```

```
// In packages/package2/index.js:
import { privateApis } from '@wordpress/package1';
import { unlock } from './lock-unlock';

const {
    privateCallback,
    privateReactComponent,
    privateClass,
    privateVariable,
} = unlock( privateApis );
```

Remember to always register the private actions and selectors on the **registered** store.

Sometimes that's easy:

```
export const store = createReduxStore( STORE_NAME, storeConfig() );
// `register` uses the same `store` object created from `createReduxStore`.
register( store );
unlock( store ).registerPrivateActions( {
    // ...
} );
```

However some package might call both `createReduxStore` **and** `registerStore`. In this case, always choose the store that gets registered:

```
export const store = createReduxStore( STORE_NAME, {
    ...storeConfig,
    persist: [ 'preferences' ],
} );
const registeredStore = registerStore( STORE_NAME, {
    ...storeConfig,
    persist: [ 'preferences' ],
} );
unlock( registeredStore ).registerPrivateActions( {
    // ...
} );
```

Private function arguments

To add a private argument to a stable function you'll need to prepare a stable and a private version of that function. Then, export the stable function and `lock()` the unstable function inside it:

```
// In @wordpress/package1/index.js:
import { lock } from './lock-unlock';

// A private function contains all the logic
function privateValidateBlocks( formula, privateIsStrict ) {
    let isValid = false;
    // ...complex logic we don't want to duplicate...
    if ( privateIsStrict ) {
        // ...
    }
    // ...complex logic we don't want to duplicate...

    return isValid;
}

// The stable public function is a thin wrapper that calls the
// private function with the private features disabled
export function validateBlocks( blocks ) {
    privateValidateBlocks( blocks, false );
}

export const privateApis = {};
lock( privateApis, { privateValidateBlocks } );
```

```
// In @wordpress/package2/index.js:
import { privateApis as package1PrivateApis } from '@wordpress/package1';
import { unlock } from './lock-unlock';

// The private function may be "unlocked" given the stable function:
const { privateValidateBlocks } = unlock( package1PrivateApis );
privateValidateBlocks( blocks, true );
```

Private React component properties

To add a private argument to a stable component you'll need to prepare a stable and a private version of that component. Then, export the stable function and `lock()` the unstable function inside it:

```
// In @wordpress/package1/index.js:
import { lock } from './lock-unlock';

// The private component contains all the logic
const PrivateMyButton = ( { title, privateShowIcon = true } ) => {
    // ...complex logic we don't want to duplicate...

    return (
        <button>
            { privateShowIcon && <Icon src={ someIcon } /> } { title }
        </button>
    );
};

// The stable public component is a thin wrapper that calls the
// private component with the private features disabled
export const MyButton = ( { title } ) => (
    <PrivateMyButton title={ title } privateShowIcon={ false } />
);

export const privateApis = {};
lock( privateApis, { PrivateMyButton } );
```

```
// In @wordpress/package2/index.js:
import { privateApis } from '@wordpress/package1';
import { unlock } from './lock-unlock';

// The private component may be "unlocked" given the stable component:
const { PrivateMyButton } = unlock( privateApis );
export function MyComponent() {
    return <PrivateMyButton data={ data } privateShowIcon={ true } />;
}
```

Private editor settings

WordPress extenders cannot update the private block settings on their own. The `updateSettings()` actions of the `@wordpress/block-editor` store will filter out all the settings that are **not** a part of the public API. The only way to actually store them is via the private action `__experimentalUpdateSettings()`.

To privatize a block editor setting, add it to the `privateSettings` list in /packages/block-editor/src/store/private-actions.ts:

```
const privateSettings = [
    'inserterMediaCategories',
    // List a block editor setting here to make it private
];
```

Private block.json and theme.json APIs

As of today, there is no way to restrict the `block.json` and `theme.json` APIs to the Gutenberg codebase. In the future, however, the new private APIs will only apply to the core WordPress blocks and plugins and themes will not be able to access them.

Inline small actions in thunks

Finally, instead of introducing a new action creator, consider using a thunk:

```
export function toggleFeature( scope, featureName ) {
    return function ( { dispatch } ) {
        dispatch( { type: '__private_BEFORE_TOGGLE' } );
        // ...
    };
}
```

Exposing private APIs publicly

Some private APIs could benefit from community feedback and it makes sense to expose them to WordPress extenders. At the same time, it doesn't make sense to turn them into a public API in WordPress core. What should you do?

You can re-export that private API as a plugin-only API to expose it publicly only in the Gutenberg plugin:

```
// This function can't be used by extenders in any context:
function privateEverywhere() {}

// This function can be used by extenders with the Gutenberg plugin but not in vanilla
WordPress Core:
function privateInCorePublicInPlugin() {}

// Gutenberg treats both functions as private APIs internally:
const privateApis = {};
lock( privateApis, { privateEverywhere, privateInCorePublicInPlugin } );

// The privateInCorePublicInPlugin function is explicitly exported,
// but this export will not be merged into WordPress core thanks to
// the globalThis.IS_GUTENBERG_PLUGIN check.
if ( globalThis.IS_GUTENBERG_PLUGIN ) {
    export const privateInCorePublicInPlugin =
        unlock( privateApis ).privateInCorePublicInPlugin;
}
```

Objects

When possible, use shorthand notation when defining object property values:

```
const a = 10;

// Bad:
const object = {
    a: a,
    performAction: function () {
        // ...
    },
};

// Good:
const object = {
    a,
    performAction() {
        // ...
```

```
    },
};
```

Strings

String literals should be declared with single-quotes *unless* the string itself contains a single-quote that would need to be escaped–in that case: use a double-quote. If the string contains a single-quote *and* a double-quote, you can use ES6 template strings to avoid escaping the quotes.

Note: The single-quote character (`'`) should never be used in place of an apostrophe (`’`) for words like `it’s` or `haven’t` in user-facing strings. For test code it’s still encouraged to use a real apostrophe.

In general, avoid backslash-escaping quotes:

```
// Bad:
const name = "Matt";
// Good:
const name = 'Matt';

// Bad:
const pet = "Matt's dog";
// Also bad (not using an apostrophe):
const pet = "Matt's dog";
// Good:
const pet = 'Matt’s dog';
// Also good:
const oddString = "She said 'This is odd.'";
```

You should use ES6 Template Strings over string concatenation whenever possible:

```
const name = 'Stacey';

// Bad:
alert( 'My name is ' + name + '.' );
// Good:
alert( `My name is ${ name }.` );
```

Optional chaining

Optional chaining is a new language feature introduced in version 2020 of the ECMAScript specification. While the feature can be very convenient for property access on objects which are potentially null-ish (`null` or `undefined`), there are a number of common pitfalls to be aware of when using optional chaining. These may be issues that linting and/or type-checking can help protect against at some point in the future. In the meantime, you will want to be cautious of the following items:

- When negating (`!`) the result of a value which is evaluated with optional chaining, you should be observant that in the case that optional chaining reaches a point where it cannot proceed, it will produce a falsy value that will be transformed to `true` when negated. In many cases, this is not an expected result.
  - Example: `const hasFocus = ! nodeRef.current?.contains( document.activeElement );` will yield `true` if `nodeRef.current` is not assigned.
  - See related issue: #21984
  - See similar ESLint rule: `no-unsafe-negation`
- When assigning a boolean value, observe that optional chaining may produce values which are falsy (`undefined`, `null`), but not strictly `false`. This can become an issue when the value is passed around in a way where it is expected to be a boolean (`true` or `false`). While it's a common occurrence for booleans—since booleans are often used in ways where the logic considers truthiness and falsyness broadly—these issues can also occur for other optional chaining when eagerly assuming a type resulting from the end of the property access chain. Type-checking may help in preventing these sorts of errors.
  - Example: `document.body.classList.toggle( 'has-focus', nodeRef.current?.contains( document.activeElement ) );` may wrongly *add* the class, since the second argument is optional. If `undefined` is passed, it would not unset the class as it would when `false` is passed.
  - Example: `<input value={ state.selected?.value.trim() } />` may inadvertently cause warnings in React by toggling between controlled and uncontrolled inputs. This is an easy trap to fall into when eagerly assuming that a result of `trim()` will always return a string value, overlooking the fact the optional chaining may have caused evaluation to abort earlier with a value of `undefined`.

React components

It is preferred to implement all components as function components, using hooks to manage component state and lifecycle. With the exception of error boundaries, you should never encounter a situation where you must use a class component. Note that the WordPress guidance on Code Refactoring applies here: There needn't be a concentrated effort to update class components in bulk. Instead, consider it as a good refactoring opportunity in combination with some other change.

JavaScript documentation using JSDoc

Gutenberg follows the WordPress JavaScript Documentation Standards, with additional guidelines relevant for its distinct use of import semantics in organizing files, and automated documentation generation using `@wordpress/docgen`.

For additional guidance, consult the JSDoc Official Documentation.

Custom types

> Prefer defining types using TypeScript syntax when possible. This guidance applies to files which have not yet been migrated to TypeScript.

Define custom types using the JSDoc `@typedef` tag.

A custom type should include a description, and should always include its base type.

Custom types should be named as succinctly as possible, while still retaining clarity of meaning and avoiding conflict with other global or scoped types. A `WP` prefix should be applied to all custom types. Avoid superfluous or redundant prefixes and suffixes (for example, a `Type` suffix, or `Custom` prefix). Custom types are not global by default, so a custom type does not need to be excessively specific to a particular package. However, they should be named with enough specificity to avoid ambiguity or name collisions when brought into the same scope as another type.

```
/**
 * A block selection object.
 *
 * @typedef WPBlockSelection
 *
 * @property {string} clientId     A block client ID.
 * @property {string} attributeKey A block attribute key.
 * @property {number} offset       An attribute value offset, based on the rich
 *                                 text value.
 */
```

Note that there is no `{Object}` between `@typedef` and the type name. As `@property`s below tells us that it is a type for objects, it is recommend to not use `{Object}` when you want to define types for your objects.

Custom types can also be used to describe a set of predefined options. While the type union can be used with literal values as an inline type, it can be difficult to align tags while still respecting a maximum line length of 80 characters. Using a custom type to define a union type can afford the opportunity to describe the purpose of these options, and helps to avoid these line length issues.

```
/**
 * Named breakpoint sizes.
 *
 * @typedef {'huge'|'wide'|'large'|'medium'|'small'|'mobile'} WPBreakpoint
 */
```

Note the use of quotes when defining a set of string literals. As in the JavaScript Coding Standards, single quotes should be used when assigning a string literal either as the type or as a default function parameter, or when specifying the path of an imported type.

Importing and exporting types

> Prefer importing and exporting types using TypeScript syntax when possible. This guidance applies to files which have not yet been migrated to TypeScript.

Use the TypeScript `import` function to import type declarations from other files or third-party dependencies.

Since an imported type declaration can occupy an excess of the available line length and become verbose when referenced multiple times, you are encouraged to create an alias of the external type using a `@typedef` declaration at the top of the file, immediately following the `import` groupings.

```
/** @typedef {import('@wordpress/data').WPDataRegistry} WPDataRegistry */
```

Note that all custom types defined in another file can be imported.

When considering which types should be made available from a WordPress package, the `@typedef` statements in the package's entry point script should be treated as effectively the same as its public API. It is important to be aware of this, both to avoid unintentionally exposing internal types on the public interface, and as a way to expose the public types of a project.

```
// packages/data/src/index.js


/** @typedef {import('./registry').WPDataRegistry} WPDataRegistry */
```

In this snippet, the `@typedef` will support the usage of the previous example's `import('@wordpress/data')`.

External dependencies

Many third-party dependencies will distribute their own TypeScript typings. For these, the `import` semantics should "just work".

Working Example: <code>import</code> type

If you use a TypeScript integration for your editor, you can typically see that this works if the type resolves to anything other than the fallback `any` type.

For packages which do not distribute their own TypeScript types, you are welcomed to install and use the DefinitelyTyped community-maintained types definitions, if one exists.

Generic types

> Prefer annotating types using TypeScript syntax when possible. This guidance applies to files which have not yet been migrated to TypeScript.

When documenting a generic type such as `Object`, `Function`, `Promise`, etc., always include details about the expected record types.

```
// Bad:


/** @type {Object} */
/** @type {Function} */
/** @type {Promise} */


// Good:


/** @type {Record<string,number>} */ /* or */ /** @type {{[setting:string]:any}} */
/** @type {(key:string)=>boolean} */
/** @type {Promise<string>} */
```

When an object is used as a dictionary, you can define its type in 2 ways: indexable interface (`{[setting:string]:any}`) or `Record`. When the name of the key for an object provides hints for developers what to do like `setting`, use indexable interface. If not, use `Record`.

The function expression here uses TypeScript's syntax for function types, which can be useful in providing more detailed information about the names and types of the expected parameters. For more information, consult the TypeScript `@type` tag function recommendations.

In more advanced cases, you may define your own custom types as a generic type using the TypeScript `@template` tag.

Similar to the "Custom Types" advice concerning type unions and with literal values, you can consider to create a custom type `@typedef` to better describe expected key values for object records, or to extract a complex function signature.

```
/**
 * An apiFetch middleware handler. Passed the fetch options, the middleware is
 * expected to call the `next` middleware once it has completed its handling.
 *
 * @typedef {(options:WPAPIFetchOptions,next:WPAPIFetchMiddleware)=>void} WPAPIFetchMiddleware
 */
```

```
/**
 * Named breakpoint sizes.
 *
 * @typedef {"huge"|"wide"|"large"|"medium"|"small"|"mobile"} WPBreakpoint
 */

/**
 * Hash of breakpoint names with pixel width at which it becomes effective.
 *
 * @type {Record<WPBreakpoint,number>}
 */
const BREAKPOINTS = { huge: 1440 /* , ... */ };
```

Nullable, undefined, and void types

> Prefer using TypeScript types when possible. This guidance applies to files which have not yet been migrated to TypeScript.

You can express a nullable type using a leading `?`. Use the nullable form of a type only if you're describing either the type or an explicit `null` value. Do not use the nullable form as an indicator of an optional parameter.

```
/**
 * Returns a configuration value for a given key, if exists. Returns null if
 * there is no configured value.
 *
 * @param {string} key Configuration key to retrieve.
 *
 * @return {?*} Configuration value, if exists.
 */
function getConfigurationValue( key ) {
    return config.hasOwnProperty( key ) ? config[ key ] : null;
}
```

Similarly, use the `undefined` type only if you're expecting an explicit value of `undefined`.

```
/**
 * Returns true if the next HTML token closes the current token.
 *
 * @param {WPHTMLToken}           currentToken Current token to compare with.
 * @param {WPHTMLToken|undefined} nextToken    Next token to compare against.
 *
 * @return {boolean} True if `nextToken` closes `currentToken`, false otherwise.
 */
```

If a parameter is optional, use the square-bracket notation. If an optional parameter has a default value which can be expressed as a default parameter in the function expression, it is not necessary to include the value in JSDoc. If the function parameter has an effective default value which requires complex logic and cannot be expressed using the default parameters syntax, you can choose to include the default value in the JSDoc.

```
/**
 * Renders a toolbar.
 *
 * @param {Object} props             Component props.
 * @param {string} [props.className] Class to set on the container `<div />`.
 */
```

When a function does not include a `return` statement, it is said to have a `void` return value. It is not necessary to include a `@return` tag if the return type is `void`.

If a function has multiple code paths where some (but not all) conditions result in a `return` statement, you can document this as a union type including the `void` type.

```
/**
 * Returns a configuration value for a given key, if exists.
 *
 * @param {string} key Configuration key to retrieve.
 *
 * @return {*|void} Configuration value, if exists.
 */
function getConfigurationValue( key ) {
    if ( config.hasOwnProperty( key ) ) {
        return config[ key ];
    }
}
```

When documenting a function type, you must always include the `void` return value type, as otherwise the function is inferred to return a mixed (“any”) value, not a void result.

```
/**
 * An apiFetch middleware handler. Passed the fetch options, the middleware is
 * expected to call the `next` middleware once it has completed its handling.
 *
 * @typedef {(options:WPAPIFetchOptions,next:WPAPIFetchMiddleware)=>void} WPAPIFetchMiddleware
 */
```

Documenting examples

Because the documentation generated using the `@wordpress/docgen` tool will include `@example` tags if they are defined, it is considered a best practice to include usage examples for functions and components. This is especially important for documented members of a package’s public API.

When documenting an example, use the markdown `\`\`\`` code block to demarcate the beginning and end of the code sample. An example can span multiple lines.

```
/**
 * Given the name of a registered store, returns an object of the store's
```

```
 * selectors. The selector functions are been pre-bound to pass the current
 * state automatically. As a consumer, you need only pass arguments of the
 * selector, if applicable.
 *
 * @param {string} name Store name.
 *
 * @example
 * ```js
 * select( 'my-shop' ).getPrice( 'hammer' );
 * ```
 *
 * @return {Record<string,WPDataSelector>} Object containing the store's
 *                                         selectors.
 */
```

Documenting React components

> Component prop types should be written using TypeScript syntax when possible. This guidance applies to files which have not yet been migrated to TypeScript.

When possible, all components should be implemented as function components, using hooks for managing component lifecycle and state.

Documenting a function component should be treated the same as any other function. The primary caveat in documenting a component is being aware that the function typically accepts only a single argument (the “props”), which may include many property members. Use the dot syntax for parameter properties to document individual prop types.

```
/**
 * Renders the block's configured title as a string, or empty if the title
 * cannot be determined.
 *
 * @example
 *
 * ```jsx
 * <BlockTitle clientId="afd1cb17-2c08-4e7a-91be-007ba7ddc3a1" />
 * ```
 *
```

```
 * @param {Object} props
 * @param {string} props.clientId Client ID of block.
 *
 * @return {?string} Block title.
 */
```

For class components, there is no recommendation for documenting the props of the component. Gutenberg does not use or endorse the `propTypes` static class member.

PHP

We use

`phpcs` (PHP_CodeSniffer) with the WordPress Coding Standards ruleset to run a lot of automated checks against all PHP code in this project. This ensures that we are consistent with WordPress PHP coding standards.

The easiest way to use PHPCS is local environment. Once that's installed, you can check your PHP by running `npm run lint:php`.

If you prefer to install PHPCS locally, you should use `composer`. Install `composer` on your computer, then run `composer install`. This will install `phpcs` and `WordPress-Coding-Standards` which you can then run via `composer lint`.

GitHub Actions workflow files

GitHub Actions workflows operate in a privileged software supply chain environment, therefore all workflow files must adhere to a high degree of quality and security standards.

These files are statically scanned when modified using Actionlint and Zizmor. Actionlint scans the YAML workflow files within the `.github/workflows` directory, while Zizmor additionally scans any action file (`action.yml`) located anywhere in the repository. It's recommended that you install both of these tools locally using a package manager to run prior to submitting changes to workflow or action files.

- GitHub Actions Workflow Standards for WordPress
- Actionlint installations instructions
- Zizmor installation instructions

To run Actionlint:

```
actionlint
```

To run Zizmor for all workflow files (note the trailing period):

```
zizmor .
```

Note: A workflow run failure will not occur when issues are detected by Zizmor. Instead, the generated report is submitted to GitHub Code Scanning and surfaced through a status check and inline annotations contextually within a pull request. Some locally reported issues may be ignored based on the repository’s configured Code Scanning settings.

□□□ #2
□ □□□ □□□ 2026-10-31 18:25:20 UTC
□ □□□ □□□ 2026-10-31 18:25:22 UTC