Backward Compatibility

- What qualifies as a production public API
  - How to preserve backward compatibility for a JavaScript function
  - How to preserve backward compatibility for a React Component
  - How to preserve backward compatibility for a Block
- Class names and DOM updates
- Deprecations
- Dev notes
  - Dev note workflow

Historically, WordPress has been known for preserving backward compatibility across versions. Gutenberg follows this example wherever possible in its production public APIs. There are rare occasions where breaking backward compatibility is unavoidable and in those cases the breakage:

- Should be constrained as much as possible to a small surface area of the API.
- Should be documented as clearly as possible to third-party developers using Dev Notes.

What qualifies as a production public API

The Gutenberg code base is composed of two different types of packages:

- **production packages**: these are packages that are shipped as WordPress scripts (example: wp-components, wp-editor...).
- **development packages**: these are made up of developer tools that can be used by third-party developers to lint, test, format and build their themes and plugins (example: @wordpress/scripts, @wordpress/env...). Typically, these are consumed as npm dependencies in third-party projects.

Backward compatibility guarantees only apply to the production packages, as updates happen through WordPress upgrades.

Production packages use the `wp` global variable to provide APIs to third-party developers. These APIs can be JavaScript functions, variables and React components.

How to preserve backward compatibility for a JavaScript function

- The name of the function should not change.
- The order of the arguments of the function should not change.
- The function's returned value type should not change.
- Changes to arguments (new arguments, modification of semantics) are possible if we guarantee that all previous calls are still possible.

How to preserve backward compatibility for a React Component

- The name of the component should not change.
- The props of the component should not be removed.
- Existing prop values should continue to be supported. If a component accepts a function as a prop, we can update the component to accept a new type for the same prop, but it shouldn't break existing usage.
- Adding new props is allowed.
- React Context dependencies can only be added or removed if we ensure the previous context contract is not breaking.

How to preserve backward compatibility for a Block

- Existing usage of the block should not break or be marked as invalid when the editor is loaded.
- The styling of the existing blocks should be guaranteed.
- Markup changes should be limited to the minimum possible, but if a block needs to change its saved markup, making previous versions invalid, a **deprecated version** of the block should be added.

Class names and DOM updates

Class names and DOM nodes used inside the tree of React components are not considered part of the public API and can be modified.

Changes to these should be done with caution as it can affect the styling and behavior of third-party code (Even if they should not rely on these in the first place). Keep the old ones if possible. If not, document the changes and write a dev note.

Deprecations

As the project evolves, flaws of existing APIs are discovered, or updates are required to support new features. When this happens, we try to guarantee that existing APIs don't break and build new alternative APIs.

To encourage third-party developers to adopt the new APIs instead, we can use the **deprecated** helper to show a message explaining the deprecation and propose the alternative whenever the old API is used.

Make it more clear when the feature was deprecated. Use the `since` and `plugin` options of the helper method.

Example:

```
deprecated( 'wp.components.ClipboardButton', {
    since: '10.3',
    plugin: 'Gutenberg',
    alternative: 'wp.compose.useCopyToClipboard',
} );
```

Dev notes

Dev notes are posts published on the make/core site prior to WordPress releases to inform third-party developers about important changes to the developer APIs, these changes can include:

- New APIs.
- Changes to existing APIs that might affect existing plugins and themes. (Example: classname changes...)
- Unavoidable backward compatibility breakage, with reasoning and migration flows.
- Important deprecations (even without breakage), with reasoning and migration flows.

Dev note workflow

- When working on a pull request and the need for a dev note is discovered, add the **Needs Dev Note** label to the PR.
- If possible, add a comment to the PR explaining why the dev note is needed.
- When the first beta of the upcoming WordPress release is shipped, go through the list of merged PRs included in the release that are tagged with the **Needs Dev Note** label.
- For each one of these PRs, write a dev note and coordinate with the WordPress release leads to publish the dev note.
- Once the dev note for a PR is published, remove the **Needs Dev Note** label from the PR.

□□□ #2
□ □□□ □□□ 2026-10-31 18:25:39 UTC
□ □□□ □□□ 2026-10-31 18:25:41 UTC