Migrating Blocks for iframe Editor Compatibility

- Overview
- What is the iframe editor?
  - Benefits of the iframe editor
  - When does the post editor work as an iframe?
  - When will the post editor work as an iframe?
- How to test your blocks in the iframe post editor
- Technical considerations for the iframe editor
  - Document and window
  - Using useRef
  - Using useRefEffect (recommended)
  - Other frameworks and libraries
  - What if the library uses global window or document and it’s out of your control?
  - Patching a library

Overview

The iframe integration is part of an ongoing effort to modernize the editing experience. The Gutenberg plugin always runs the post editor inside an iframe, building upon the original iframe migration in the template editor. WordPress Core is moving toward the same behavior.

This guide encourages migration of blocks to API version 3 in preparation for the planned iframe integration of the post editor. It helps verify in advance that blocks work in the iframe editor and assists in updating blocks so they work correctly in the iframe environment.

What is the iframe editor?

Benefits of the iframe editor

From a technical perspective, the iframe editor provides several important benefits:

- **Style isolation**: Admin styles no longer affect the editor content, eliminating the need to reset admin CSS rules. Content styles no longer affect the admin screen, so block and theme CSS rules no longer need to be prefixed.
- **Viewport-relative units**: Viewport-relative CSS units (`vw`, `vh`) work correctly. The dimensions of the editor content are usually not the same as the dimensions of the admin page, so without an iframe, units like `vw` would be relative to the admin page.
- **Media queries**: Media queries work natively without needing workarounds, which were fragile.
- **Easier development**: Block and theme authors benefit because styles from the front-end can be dropped in with very little, if anything, to adjust. This also applies to lighter blocks, where the editor DOM structure matches the front-end.
- **Selection handling**: With a separate window for the editor content, it's possible for the selection in the editor to remain visible while also having a (collapsed) selection in the editor UI, for example an input field for a URL.

The iframed post editor will make life easier for block and theme authors by reducing styling conflicts and improving layout accuracy.

When does the post editor work as an iframe?

While most editors, including the template editor, already work as iframes, WordPress 7.0 retains a conditional fallback for backward compatibility (determined by the useShouldIframe hook):

- **If the Gutenberg plugin is enabled:** The editor always works as an iframe.
- **If the Gutenberg plugin is not enabled:** The editor works as an iframe when any of the following is true: the device type is not `Desktop` (for example, tablet or mobile previews), the current post type is `wp_template` or `wp_block`, zoom-out mode is active, or all blocks present in the post content have `apiVersion` 3 or higher.

In WordPress 7.0 without the Gutenberg plugin, keeping a block at `apiVersion` 2 can trigger the non-iframe fallback when that block is present in the post. This is only a temporary compatibility path: it does not apply when the Gutenberg plugin is enabled and is planned for removal in WordPress 7.1. Blocks should migrate to `apiVersion` 3 after they have been tested in the iframe

editor.

When will the post editor work as an iframe?

In Gutenberg 23.6 and WordPress 7.1, the post editor always works as an iframe, regardless of the `apiVersion` of the blocks in the post content. The conditional fallback to a non-iframe editor is removed, so blocks with `apiVersion` 2 or lower always run inside the iframe.

Ahead of this, to encourage developers to test in the iframe editor, WordPress 6.9 introduces a browser console warning when blocks are registered with `apiVersion` 2 or lower, and updates the block.json schema to only allow `apiVersion: 3`. For details, see Preparing the post editor for full iframe integration.

In WordPress 7.0, the iframe condition is evaluated against the `apiVersion` of *blocks actually inserted in the post content*, instead of *all registered blocks*. The iframe is still not enforced: if a block with `apiVersion` 2 or lower is in the content, the post editor falls back to a non-iframe editor. For details, see Iframed editor changes in WordPress 7.0.

How to test your blocks in the iframe post editor

All core blocks already use `apiVersion` 3. After testing your block in the iframe editor, update it to `apiVersion` 3.

The simplest way to test is with Gutenberg 23.6 or later, where the post editor always uses the iframe. In WordPress 7.0 without the Gutenberg plugin, the editor may fall back to a non-iframe editor when the post contains a block using `apiVersion` 2 or lower.

Technical considerations for the iframe editor

Most blocks should work in the iframe editor without modification, but the following technical considerations and things to be aware of are documented below.

Document and window

The iframe will have a different `document` and `window` than the admin page, which is now the parent window. Editor scripts are loaded in the admin page, so accessing the `document` or `window` to do something with the content will no longer work.

Most blocks written in React should continue to work properly, except if you rely on `document` or `window`. To fix, you need to create a ref to access the relative document (`ownerDocument`) or window (`defaultView`). Regardless of the iframe, it is good practice to do this and avoid the use of globals.

Using useRef

```
import { __ } from '@wordpress/i18n';
import { useBlockProps } from '@wordpress/block-editor';
import { useRef, useEffect } from '@wordpress/element';

export default function Edit() {
    const ref = useRef();

    useEffect( () => {
        const { ownerDocument } = ref.current;
        const { defaultView } = ownerDocument;
        defaultView.addEventListener( ... );
        return () => {
            defaultView.removeEventListener( ... );
        };
    }, [] );

    const blockProps = useBlockProps( { ref } );

    return (
        <div { ...blockProps }>
            Hello world!
        </div>
    );
}
```

Using useRefEffect (recommended)

If you attach event handlers, remember that the `useEffect` callback will not be called if the ref changes, so it is good practice to use the new `useRefEffect` API, which *will* call the given callback if the ref changes in addition to any dependencies passed.

```
import { __ } from '@wordpress/i18n';
import { useBlockProps } from '@wordpress/block-editor';
import { useRefEffect } from '@wordpress/compose';

export default function Edit() {
    const ref = useRefEffect( ( element ) => {
        const { ownerDocument } = element;
        const { defaultView } = ownerDocument;
        defaultView.addEventListener( ... );
        return () => {
            defaultView.removeEventListener( ... );
        };
    }, [] );

    const blockProps = useBlockProps( { ref } );

    return (
        <div { ...blockProps }>
            Hello world!
        </div>
    );
}
```

Other frameworks and libraries

For the editor, scripts such as jQuery are loaded in the parent window (admin page), which is fine. When using these to interact with a block in the iframe, you should pass the element reference.

```
import { __ } from '@wordpress/i18n';
import { useBlockProps } from '@wordpress/block-editor';
import { useRefEffect } from '@wordpress/compose';
import jQuery from 'jquery';

export default function Edit() {
    const ref = useRefEffect( ( element ) => {
```

```
        jQuery( element ).masonry( … );
        return () => {
            jQuery( element ).masonry( 'destroy' );
        }
    }, [] );

    const blockProps = useBlockProps( { ref } );

    return (
        <div { ...blockProps }>
            Hello world!
        </div>
    );
}
```

What if the library uses global window or document and it’s out of your control?

Submit an issue or PR for the library to use `ownerDocument` and `defaultView` instead of the globals. Ideally, any library should allow initialization with an element in an iframe as the target. It’s never impossible. Feel free to contact us to mention the issue.

In the meantime, you can use the script that is loaded inside the iframe. We’ve loaded all front-end scripts in the iframe to fix these cases, but note that ideally you shouldn’t use scripts loaded in the iframe at all. You can use `defaultView` to access the script.

```
import { __ } from '@wordpress/i18n';
import { useBlockProps } from '@wordpress/block-editor';
import { useRefEffect } from '@wordpress/compose';
import jQuery from 'jquery';

export default function Edit() {
    const ref = useRefEffect( ( element ) => {
        const { ownerDocument } = element;
        const { defaultView } = ownerDocument;

        // Use the script loaded in the iframe.
        // Scripts are loaded asynchronously, so check if the script is loaded.
        // After the dependencies have loaded, the block will re-render.
```

```
        if ( ! defaultView.jQuery ) {
            return;
        }

        defaultView.jQuery( element ).masonry( … );
        return () => {
            defaultView.jQuery( element ).masonry( 'destroy' );
        }
    } );

    const blockProps = useBlockProps( { ref } );

    return (
        <div { ...blockProps }>
            Hello world!
        </div>
    );
}
```

Patching a library

When an external library references the global `document` or `window` and you cannot wait for an upstream fix, you can apply your own patch to make it iframe compatible. The following steps use patch-package to keep the patch under version control (assuming npm):

1. Edit the library's code directly in `node_modules` to make it work, replacing global `document` / `window` references with `ownerDocument` / `defaultView` derived from the block's element. For example, for `@panzoom/panzoom`, edit `node_modules/@panzoom/panzoom/dist/panzoom.es.js`.
2. Install `patch-package`:
   `bash`
   `npm add -D patch-package`
3. Generate the patch for the library:
   `bash`
   `npm exec --no -- patch-package @panzoom/panzoom`
4. Add a `postinstall` script to your `package.json` so the patch is applied automatically after every install:
   `json`
   `"scripts": {`
   `"postinstall": "patch-package"`
   `}`
5. Commit the generated `.patch` file (in the `patches` directory) and the updated `package.json` to version control.

The patch typically replaces global references with the element's `ownerDocument`. For example, the patch for `@panzoom/panzoom` looks like this:

```
 function isAttached(node) {
+    var { ownerDocument } = node;
     var currentNode = node;
     while (currentNode && currentNode.parentNode) {
-        if (currentNode.parentNode === document)
+        if (currentNode.parentNode === ownerDocument)
             return true;
         currentNode =
             currentNode.parentNode instanceof ShadowRoot
@@ function Panzoom(elem, options) {
         bound = true;
+        var { ownerDocument } = elem;
         onPointer('down', options.canvas ? parent : elem, handleDown);
-        onPointer('move', document, handleMove, { passive: true });
-        onPointer('up', document, handleUp, { passive: true });
+        onPointer('move', ownerDocument, handleMove, { passive: true });
+        onPointer('up', ownerDocument, handleUp, { passive: true });
     }
```

Even when patching works, prefer submitting an upstream issue or PR so the library supports iframe environments natively.

□□□ #2
□ □□□ □□□ 2026-10-31 18:26:18 UTC
□ □□□ □□□ 2026-10-31 18:26:20 UTC