

Block Editor Handbook

In this article

Table of Contents

- [Navigating this handbook](#)
- [Further resources](#)
- [Are you in the right place?](#)

[↑ Back to top](#)

Welcome to the Block Editor Handbook.

The **Block Editor** is a modern and up-to-date paradigm for WordPress site building and publishing. It uses a modular system of **Blocks** to compose and format content and is designed to create rich and flexible layouts for websites and digital products.

The editor consists of several primary elements, as shown in the following figure:



X



Search



Blocks

Patterns

Media

TEXT



Paragraph

Heading

List



Quote

Classic

Code



Preformatted

Pullquote

Table

The elements highlighted in the figure are:

1. **Inserter**: A panel for inserting blocks into the content canvas
2. **Content canvas**: The content editor, which holds content created with blocks
3. **Settings Sidebar**: A sidebar panel for configuring a block's settings (among other things)

Through the Block editor, you create content modularly using Blocks. There are many [core blocks](#) ready to be used, and you can also [create your own custom block](#).

A [Block](#) is a discrete element such as a Paragraph, Heading, Media, or Embed. Each block is treated as a separate element with individual editing and format controls. When all these components are pieced together, they make up the content that is then [stored in the WordPress database](#).

The Block Editor is the result of the work done on the [Gutenberg project](#), which aims to revolutionize the WordPress editing experience.

Besides offering an [enhanced editing experience](#) through visual content creation tools, the Block Editor is also a powerful developer platform with a [rich feature set of APIs](#) that allow it to be manipulated and extended many different ways.

Navigating this handbook

This handbook is focused on block development and is divided into five sections, each serving a different purpose.

- **Getting Started** – For those just starting out with block development, this is where you can get set up with a [development environment](#) and learn the [fundamentals of block development](#). Its [Quick Start Guide](#) and [Tutorial: Build your first block](#) are probably the best places to start learning block development.
- **How-to Guides** – Here, you can build on what you learned in the Getting Started section and learn how to solve particular problems you might encounter. You can also get tutorials and example code that you can reuse for projects such as [working with WordPress' data](#) or [Curating the Editor Experience](#).
- **Reference Guides** – This section is the heart of the handbook and is where you can get down to the nitty-gritty and look up the details of the particular API you're working with or need information on. Among other things, the [Block API Reference](#) covers most of what you will want to do with a block, and each [component](#) and [package](#) is also documented here. *Components are also documented via [Storybook](#).*
- **Explanations** – This section enables you to go deeper and reinforce your practical knowledge with a theoretical understanding of the [Architecture](#) of the block editor.
- **Contributor Guide** – Gutenberg is open source software, and anyone is welcome to contribute to the project. This section details how to contribute and can help you choose in which way you want to contribute, whether with [code](#), [design](#), [documentation](#), or in some other way.

Further resources

This handbook should be considered the canonical resource for all things related to block development. However, there are other resources that can help you.

- [**WordPress Developer Blog**](#) – An ever-growing resource of technical articles covering specific topics related to block development and a wide variety of use cases. The blog is also an excellent way to [keep up with the latest developments in WordPress](#).
- [**Learn WordPress**](#) – The WordPress hub for learning resources where you can find courses like [Introduction to Block Development: Build your first custom block](#), [Converting a Shortcode to a Block](#) or [Using the WordPress Data Layer](#)
- [**WordPress.tv**](#) – A hub of WordPress-related videos (from talks at WordCamps to recordings of online workshops) curated and moderated by the WordPress.org community. You’re sure to find something to aid your learning about [block development](#) or the [Block Editor](#) here.
- [**Gutenberg repository**](#) – Development of the block editor project is carried out in this GitHub repository. It contains the code of interesting packages such as [block-library](#) (core blocks) or [components](#) (common UI elements). *The [block-development-examples](#) repository is another useful reference.*
- [**End User Documentation**](#) – This documentation site is targeted to the end user (not developers), where you can also find documentation about the [Block Editor](#) and [working with blocks](#).

Are you in the right place?

[This handbook](#) is targeted at those seeking to develop for the block editor, but several other handbooks exist for WordPress developers under [developer.wordpress.org](#):

- [/themes](#) – Theme Handbook
- [/plugins](#) – Plugin Handbook
- [/apis](#) – Common APIs Handbook
- [/advanced-administration](#) – WP Advanced Administration Handbook
- [/rest-api](#) – REST API Handbook
- [/coding-standards](#) – Best practices for WordPress developers

First published

May 2, 2019

Last updated

January 19, 2024

Edit article

[Improve it on GitHub: Block Editor Handbook”](#)

[Next](#) [Getting Started](#) [Next: Getting Started](#)

Getting Started

In this article

Table of Contents

- [Navigating this chapter](#)
- [Getting Started on the WordPress project and Gutenberg](#)
 - [Ways to Stay Informed](#)
- [Additional Resources](#)

[↑ Back to top](#)

Welcome! Let's get started building with blocks. Blocks are at the core of extending WordPress. You can create custom blocks, your own block patterns, or combine them together to build a block theme.

[**Navigating this chapter**](#)

For those starting with block development, this section is the perfect starting point as it provides the knowledge you need to start creating your own custom blocks.

- [**Block Development Environment**](#) – Set up the right development environment to create blocks and get introduced to basic tools for block development such as [wp-env](#), [create-block](#) and [wp-scripts](#)
- [**Quick Start Guide**](#) – Get a block up and running in less than 1 min.
- [**Tutorial: Build your first block**](#) – The tutorial will guide you, step by step, through the complete process of creating a fully functional custom block.
- [**Fundamentals of Block Development**](#) – This section provides an introduction to the most relevant concepts in Block Development.
- [**Glossary**](#) – Glossary of terms related to the Block Editor and Full Site Editing
- [**Frequently Asked Questions**](#) – Set of questions (and answers) that have come up from the last few years of Gutenberg development.

[**Getting Started on the WordPress project and Gutenberg**](#)

At a high level, here are a few ways to begin your journey but read on to explore more:

- Learn more about where this work is going by [reviewing the long term roadmap](#).
- Explore the [GitHub repo](#) to see the latest issues and PRs folks are working on, especially [Good First Issues](#).
- Join the [Slack community](#) to join meetings, ongoing conversations, and more.
- Take courses on how to use the block editor and more on [Learn WordPress](#).
- Expand your knowledge by reviewing more developer docs at the overall [developer.wordpress.org resource](#).
- Subscribe to [updates on Make Core](#), the main site where ongoing project updates happen.

Ways to Stay Informed

New features and changes are important to keep up to date on as the Gutenberg project continues. Each person will have their own unique needs in keeping up with a project of this scale. What follows is more of a catalogue of ways to keep up rather than a recommendation for how to do so.

- [Keeping up with Gutenberg](#) – compilation of Gutenberg-related team posts of Core, Core-Editor, Core-js, Core-css, Design, Meta, and Themes, and other teams.
- [“What’s New In Gutenberg?” release posts](#). These updates are wrangled by the Core Editor team and focus on what’s been released in each biweekly Gutenberg release. They include the most relevant features released and a full changelog.
- [Core Editor meetings](#). These meetings are wrangled by volunteer members in the #core-editor Slack channel. [Agendas](#) and [summaries](#) are shared on the [Make Core blog](#). They focus on task coordination and relevant discussions around Gutenberg releases. There is an Open Floor period in each chat where people can suggest topics to discuss.
- Checking in on [issues](#) and [PRs](#) on GitHub. This will give you a nearly real-time understanding of what’s being worked on by the developers and designers.

Additional Resources

The [block-development-examples](#) repo is the central hub of examples for block development referenced from this handbook.

At [Learn WordPress](#), you can find [tutorials](#), [courses](#), and [online workshops](#) to learn more about developing for the Block Editor. Here is a selection of current offerings:

- [Intro to Block Development: Build Your First Custom Block](#)
- [Converting a Shortcode to a Block](#)
- [Using the WordPress Data Layer](#)
- [Registering Block Patterns](#)
- [Intro to Gutenberg Block Development](#)
- [Intro to Publishing with the Block Editor](#)

First published

October 22, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Getting Started”](#)

[Next Block Development Environment](#) [Next: Block Development Environment](#)

Block Development Environment

In this article

Table of Contents

- [Code editor](#)
- [Node.js development tools](#)
- [Local WordPress environment](#)

[↑ Back to top](#)

This guide will help you set up the right development environment to create blocks and other plugins that extend and modify the Block Editor in WordPress.

To contribute to the Gutenberg project itself, refer to the additional documentation in the [code contribution guide](#).

A block development environment includes the tools you need on your computer to successfully develop for the Block Editor. The three essential requirements are:

1. [Code editor](#)
2. [Node.js development tools](#)
3. [Local WordPress environment \(site\)](#)

[Code editor](#)

A code editor is used to write code, and you can use whichever editor you're most comfortable with. The key is having a way to open, edit, and save text files.

If you do not already have a preferred code editor, [Visual Studio Code](#) (VS Code) is a popular choice for JavaScript development among Core contributors. It works well across the three major platforms (Windows, Linux, and Mac), is open-source, and is actively maintained by Microsoft. VS Code also has a vibrant community providing plugins and extensions, including many for WordPress development.

[Node.js development tools](#)

Node.js (node) is an open-source runtime environment that allows you to execute JavaScript outside of the web browser. While Node.js is not required for all WordPress JavaScript development, it's essential when working with modern JavaScript tools and developing for the Block Editor.

Node.js and its accompanying development tools allow you to:

- Install and run WordPress packages needed for Block Editor development, such as `wp-scripts`
- Setup local WordPress environments with `wp-env` and `wp-now`
- Use the latest ECMAScript features and write code in ESNext
- Lint, format, and test JavaScript code

- Scaffold custom blocks with the `create-block` package

The list goes on. While modern JavaScript development can be challenging, WordPress provides several tools, like [wp-scripts](#) and [create-block](#), that streamline the process and are made possible by Node.js development tools.

The recommended Node.js version for block development is [Active LTS \(Long Term Support\)](#). However, there are times when you need to use different versions. A Node.js version manager tool like `nvm` is strongly recommended and allows you to easily change your node version when required. You will also need Node Package Manager (`npm`) and the Node Package eXecute (`npx`) to work with some WordPress packages. Both are installed automatically with Node.js.

To be able to use the Node.js tools and [packages provided by WordPress](#) for block development, you'll need to set a proper Node.js runtime environment on your machine. To learn more about how to do this, refer to the links below.

- [Install Node.js for Mac and Linux](#)
- [Install Node.js for Windows](#)

[Local WordPress environment](#)

A local WordPress environment (site) provides a controlled, efficient, and secure space for development, allowing you to build and test your code before deploying it to a production site. The [same requirements](#) for WordPress apply to local sites.

In the broader WordPress community, there are many available tools for setting up a local WordPress environment on your computer. The Block Editor Handbook covers `wp-env`, which is open-source and maintained by the WordPress project itself. It's also the recommended tool for Gutenberg development.

Refer to the [Get started with wp-env](#) guide for setup instructions.

Throughout the Handbook, you may also see references to [wp-now](#). This is a lightweight tool powered by WordPress Playground that streamlines setting up a simple local WordPress environment. While still experimental, this tool is great for quickly testing WordPress releases, plugins, and themes.

This list is not exhaustive, but here are several additional options to choose from if you prefer not to use `wp-env`:

- [Local](#)
- [XAMPP](#)
- [MAMP](#)
- [Varying Vagrant Vagrants \(VVV\)](#)

First published

March 9, 2021

Last updated

January 16, 2024

Edit article

Node.js development environment

In this article

Table of Contents

- [Node.js installation on Mac and Linux \(with nvm\)](#)
- [Node.js installation on Windows and others](#)
- [Troubleshooting](#)
- [Additional resources](#)

[↑ Back to top](#)

When developing for the Block Editor, you will need [Node.js](#) development tools along with a code editor and a local WordPress environment (see [Block Development Environment](#)). Node.js (`node`) is an open-source runtime environment that allows you to execute JavaScript code from the terminal (also known as a command-line interface, CLI, or shell).

Installing `node` will automatically include the Node Package Manager (`npm`) and the Node Package eXecute (`npx`), two tools you will frequently use in block and plugin development.

Node Package Manager ([npm](#)) serves multiple purposes, including dependency management and script execution. It's the recommended package manager and is extensively featured in all documentation.

The Node Package eXecute ([npx](#)) tool is used to run commands from packages without installing them globally and is commonly used when scaffolding blocks with the [create-block](#) package.

[Node.js installation on Mac and Linux \(with nvm\)](#)

It's recommended that you use [Node Version Manager](#) (`nvm`) to install Node.js. This allows you to install and manage specific versions of `node`, which are installed locally in your home directory, avoiding any global permission issues.

Here are the quick instructions for installing `node` using `nvm` and setting the recommended Node.js version for block development. See the [complete installation guide](#) for more details.

1. Open the terminal and run the following to install `nvm`. On macOS, the required developer tools are not installed by default. Install them if prompted.

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh |
```

1. Quit and restart the terminal.
2. Run `nvm install --lts` in the terminal to install the latest [LTS](#) (Long Term Support) version of Node.js.

3. Run `node -v` and `npm -v` in the terminal to verify the installed node and npm versions.

If needed, you can also install specific versions of node. For example, install version 18 by running `nvm install 18`, and switch between different versions by running `nvm use [version-number]`. See the [nvm usage guide](#) for more details.

Some projects, like Gutenberg, include an [.nvmrc](#) file which specifies the version of node that should be used. In this case, running `nvm use` will automatically select the correct version. If the version is not yet installed, you will get an error that tells you what version needs to be added. Run `nvm install [version-number]` followed by `nvm use`.

[Node.js installation on Windows and others](#)

You can [download a Node.js installer](#) directly from the main Node.js website. The latest version is recommended. Installers are available for Windows and Mac, and binaries are available for Linux.

Microsoft also provides a [detailed guide](#) on how to install nvm and Node.js on Windows and WSL.

[Troubleshooting](#)

If you encounter the error `zsh: command not found: nvm` when attempting to install node, you might need to create the default profile file.

The default shell is `zsh` on macOS, so create the profile file by running `touch ~/.zshrc` in the terminal. It's fine to run if the file already exists. The default profile is `bash` for Ubuntu, including WSL, so use `touch ~/.bashrc` instead. Then repeat steps 2-4.

The latest node version should work for most development projects, but be aware that some packages and tools have specific requirements. If you encounter issues, you might need to install and use a previous node version. Also, make sure to check if the project has an `.nvmrc` and use the node version indicated.

[Additional resources](#)

- [Node.js](#) (Official documentation)
- [Node Version Manager](#) (Official documentation)
- [Installing Node.js and npm for local WordPress development](#) (Learn WordPress tutorial)

First published

September 18, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Node.js development environment”](#)

Get started with wp-env

In this article

Table of Contents

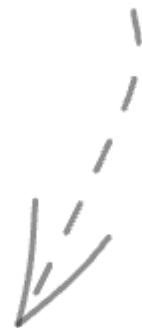
- [Quick start](#)
- [Set up Docker Desktop](#)
- [Install and run wp-env](#)
 - [Where to run wp-env](#)
 - [Uninstall or reset wp-env](#)
- [Troubleshooting](#)
 - [Common errors](#)
 - [Ubuntu Docker setup](#)
- [Additional resources](#)

[↑ Back to top](#)

The [@wordpress/env](#) package (`wp-env`) lets you set up a local WordPress environment (site) for building and testing plugins and themes, without any additional configuration.

Before following this guide, install [Node.js development tools](#) if you have not already done so.

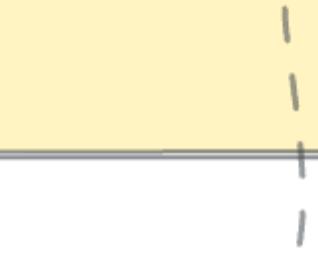
if run from a plugin or theme
it will launch a WordPress
with that plugin or theme in



wp-env can be run from anywhere

```
npx @wordpress/env start
```

npx allows you to run a command
from a remote npm package
(without installing it)



Quick start

1. Download, install, and start [Docker Desktop](#) following the instructions for your operating system.
2. Run `npm -g install @wordpress/env` in the terminal to install `wp-env` globally.
3. In the terminal, navigate to an existing plugin directory, theme directory, or a new working directory.
4. Run `wp-env start` in the terminal to start the local WordPress environment.
5. After the script runs, navigate to `http://localhost:8888/wp-admin` and log into the WordPress dashboard using username `admin` and password `password`.

Set up Docker Desktop

The `wp-env` tool uses [Docker](#) to create a virtual machine that runs the local WordPress site. The Docker Desktop application is free for small businesses, personal use, education, and non-commercial open-source projects. See their [FAQ](#) for more information.

Use the links below to download and install Docker Desktop for your operating system.

- [Docker Desktop for Mac](#)
- [Docker Desktop for Windows](#)
- [Docker Desktop for Linux](#)

If you are using a version of Ubuntu prior to 20.04.1, see the additional [troubleshooting notes](#) below.

After successful installation, start the Docker Desktop application and follow the prompts to get set up. You should generally use the recommended (default) settings, and creating a Docker account is optional.

Install and run wp-env

The `wp-env` tool is used to create a local WordPress environment with Docker. So, after you have set up Docker Desktop, open the terminal and install the `wp-env` by running the command:

```
npm -g install @wordpress/env
```

This will install the `wp-env` globally, allowing the tool to be run from any directory. To confirm it's installed and available, run `wp-env --version`, and the version number should appear.

Next, navigate to an existing plugin directory, theme directory, or a new working directory in the terminal and run:

```
wp-env start
```

Once the script completes, you can access the local environment at: `http://localhost:8888`. Log into the WordPress dashboard using username `admin` and password `password`.

Some projects, like Gutenberg, include their own specific `wp-env` configurations, and the documentation might prompt you to run `npm run start wp-env` instead.

For more information on controlling the Docker environment, see the [@wordpress/env package](#) readme.

Where to run wp-env

The `wp-env` tool can run from practically anywhere. When using the script while developing a single plugin, `wp-env start` can mount and activate the plugin automatically when run from the directory containing the plugin. This also works for themes when run from the directory in which you are developing the theme.

A generic WordPress environment will be created if you run `wp-env start` from a directory that is not a plugin or theme. The script will display the following warning, but ignore if this is your intention.

```
!! Warning: could not find a .wp-env.json configuration file and could not
```

You can also use the `.wp-env.json` configuration file to create an environment that works with multiple plugins and/or themes. See the [@wordpress/env package](#) readme for more configuration details.

Uninstall or reset wp-env

Here are a few instructions if you need to start over or want to remove what was installed.

- If you just want to reset and clean the WordPress database, run `wp-env clean all`
- To remove the local environment completely for a specific project, run `wp-env destroy`
- To globally uninstall the `wp-env` tool, run `npm -g uninstall @wordpress/env`

Troubleshooting

Common errors

When using `wp-env`, it's common to get the error: `Error while running docker-compose command`

- Check that Docker Desktop is started and running.
- Check Docker Desktop dashboard for logs, restart, or remove existing virtual machines.
- Then try rerunning `wp-env start`.

If you see the error: `Host is already in use by another container`

- The container you are attempting to start is already running, or another container is. You can stop an existing container by running `wp-env stop` from the directory that you started it in.
- If you do not remember the directory where you started `wp-env`, you can stop all containers by running `docker stop $(docker ps -q)`. This will stop all Docker containers, so use with caution.
- Then try rerunning `wp-env start`.

Ubuntu Docker setup

If you are using a version of Ubuntu prior to 20.04.1, you may encounter errors when setting up a local WordPress environment with `wp-env`.

To resolve this, start by following the [installation guide](#) from Docker. docker-compose is also required, which you may need to install separately. Refer to the [Docker compose documentation](#).

Once Docker and wp-env are installed, and assuming wp-env is configured globally, try running `wp-env start` in a directory. If you run into this error:

```
ERROR: Couldn't connect to Docker daemon at http+docker://localhost - is it running?
```

If it's at a non-standard location, specify the URL with the `DOCKER_HOST` environment variable:

```
/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

If Docker is not running, try starting the service by running `sudo systemctl start docker.service`.

If Docker is running, then it is not listening to how the WordPress environment is trying to communicate. Try adding the following service override file to include listening on `tcp`. See [this Docker documentation](#) on how to configure remote access for Docker daemon.

```
# /etc/systemd/system/docker.service.d/override.conf
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376
```

Restart the service from the command-line:

```
sudo systemctl daemon-reload
sudo systemctl restart docker.service
```

After restarting the services, set the environment variable `DOCKER_HOST` and try starting `wp-env` with:

```
export DOCKER_HOST=tcp://127.0.0.1:2376
wp-env start
```

Your environment should now be set up at `http://localhost:8888`.

Additional resources

- [@wordpress/env](#) (Official documentation)
- [Docker Desktop](#) (Official documentation)
- [Quick and easy local WordPress development with wp-env](#) (WordPress Developer Blog)
- [wp-env: Simple Local Environments for WordPress](#) (Make WordPress Core Blog)
- [wp-env Basics diagram](#) (Excalidraw)

First published

September 18, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Get started with wp-env”](#)

[Previous Node.js development environment](#) [Previous: Node.js development environment](#)

[Next Get started with create-block](#) [Next: Get started with create-block](#)

Get started with create-block

In this article

[Table of Contents](#)

- [Quick start](#)
 - [Installation](#)
 - [Basic usage](#)
- [Alternate implementations](#)
 - [Interactive mode](#)
 - [Quick start mode using options](#)
 - [Using templates](#)
- [Additional resources](#)

[↑ Back to top](#)

Custom blocks for the Block Editor in WordPress are typically registered using plugins and are defined through a specific set of files. The [@wordpress/create-block](#) package is an officially supported tool to scaffold the structure of files needed to create and register a block. It generates all the necessary code to start a project and integrates a modern JavaScript build setup (using [wp-scripts](#)) with no configuration required.

The package is designed to help developers quickly set up a block development environment following WordPress best practices.

[Quick start](#)

[Installation](#)

Start by ensuring you have Node.js and npm installed on your computer. Review the [Node.js development environment](#) guide if not.

You can use `create-block` to scaffold a block just about anywhere and then [use wp-env](#) from the inside of the generated plugin folder. This will create a local WordPress development environment with your new block plugin installed and activated.

If you have your own [local WordPress development environment](#) already set up, navigate to the `plugins/` folder using the terminal.

Run the following command to scaffold an example block plugin:

```
npx @wordpress/create-block@latest todo-list  
cd todo-list
```

The `slug` provided (`todo-list`) defines the folder name for the scaffolded plugin and the internal block name.

Navigate to the Plugins page of our local WordPress installation and activate the “Todo List” plugin. The example block will then be available in the Editor.

[Basic usage](#)

The `create-block` assumes you will use modern JavaScript (ESNext and JSX) to build your block. This requires a build step to compile the code into a format that browsers can understand. Luckily, the `wp-scripts` package handles this process for you. Refer to the [Get started with wp-scripts](#) for an introduction to this package.

When `create-block` scaffolds the block, it installs `wp-scripts` and adds the most common scripts to the block’s `package.json` file. By default, those include:

```
{  
  "scripts": {  
    "build": "wp-scripts build",  
    "format": "wp-scripts format",  
    "lint:css": "wp-scripts lint-style",  
    "lint:js": "wp-scripts lint-js",  
    "packages-update": "wp-scripts packages-update",  
    "plugin-zip": "wp-scripts plugin-zip",  
    "start": "wp-scripts start"  
  }  
}
```

These scripts can then be run using the command `npm run {script name}`. The two scripts you will use most often are `start` and `build` since they handle the build step.

When working on your block, use the `npm run start` command. This will start a development server and automatically rebuild the block whenever any code change is detected.

When you are ready to deploy your block, use the `npm run build` command. This optimizes your code and makes it production-ready.

See the `wp-scripts` [package documentation](#) for more details about each available script.

[Alternate implementations](#)

[Interactive mode](#)

For developers who prefer a more guided experience, the `create-block` package provides an interactive mode. Instead of manually specifying all options upfront, like the `slug` in the above example, this mode will prompt you for inputs step-by-step.

To use this mode, run the command:

```
npx @wordpress/create-block@latest
```

Follow the prompts to configure your block settings interactively.

Quick start mode using options

If you’re already familiar with the `create-block` [options](#) and want a more streamlined setup, you can use quick start mode. This allows you to pass specific options directly in the command line, eliminating the need for interactive prompts.

For instance, to quickly create a block named “my-block” with a namespace of “my-plugin” that is a Dynamic block, use this command:

```
npx @wordpress/create-block@latest --namespace="my-plugin" --slug="my-bloc
```

Using templates

The `create-block` package also supports the use of templates, enabling you to create blocks based on predefined configurations and structures. This is especially useful when you have a preferred block structure or when you’re building multiple blocks with similar configurations.

To use a template, specify the `--template` option followed by the template name or path:

```
npx @wordpress/create-block --template="my-custom-template"
```

Templates must be set up in advance so the `create-block` package knows where to find them. Learn more in the `create-block` [documentation](#), and review the [External Project Templates](#) guide.

Additional resources

- [Using the create-block tool](#) (Learn WordPress tutorial)
- [@wordpress/create-block](#) (Official documentation)
- [@wordpress/scripts](#) (Official documentation)

First published

October 17, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Get started with create-block”](#)

[Previous](#) [Get started with wp-env](#) [Previous: Get started with wp-env](#)
[Next](#) [Get started with wp-scripts](#) [Next: Get started with wp-scripts](#)

Get started with wp-scripts

In this article

Table of Contents

- [Quick start](#)
 - [Installation](#)
 - [Basic usage](#)
 - [The build process with wp-scripts](#)
 - [Enqueuing assets](#)
- [Next steps](#)
 - [Maintaining code quality](#)
 - [Running tests](#)
 - [Advanced configurations](#)
- [Additional resources](#)

[↑ Back to top](#)

The [@wordpress/scripts](#) package, commonly referred to as `wp-scripts`, is a set of configuration files and scripts that primarily aims to standardize and simplify the development process of WordPress projects that require a JavaScript build step.

A JavaScript build step refers to the process of transforming, bundling, and optimizing JavaScript source code and related assets into a format suitable for production environments. These build steps often take modern JavaScript (ESNext and JSX) and convert it to a version compatible with most browsers. They can also bundle multiple files into one, minify the code to reduce file size and perform various other tasks to optimize the code.

You will typically be working with ESNext and JSX when building for the Block Editor, and most examples in the Block Editor Handbook are written in these syntaxes. Learning how to set up a build step is essential. However, configuring the necessary tools like [webpack](#), [Babel](#), and [ESLint](#) can become complex. This is where `wp-scripts` comes in.

Here are a few things that `wp-scripts` can do:

- **Compilation:** Converts modern JavaScript (ESNext and JSX) into code compatible with most browsers, using Babel.
- **Bundling:** Uses webpack to combine multiple JavaScript files into a single bundle for better performance.
- **Code Linting:** Provides configurations for ESLint to help ensure code quality and conformity to coding standards.
- **Code Formatting:** Incorporates Prettier for automated code styling to maintain consistent code formatting across projects.
- **Sass Compilation:** Converts Sass (.scss or .sass) files to standard CSS.
- **Code Minification:** Reduces the size of the JavaScript code for production to ensure faster page loads.

The package abstracts away much of the initial setup, configuration, and boilerplate code associated with JavaScript development for modern WordPress. You can then focus on building blocks and Block Editor extensions.

Quick start

If you use [@wordpress/create-block](#) package to scaffold the structure of files needed to create and register a block, you'll also get a modern JavaScript build setup (using `wp-scripts`) with no configuration required, so you don't need to worry about installing `wp-scripts` or enqueueing assets. Refer to [Get started with create-block](#) for more details.

Installation

Ensure you have Node.js and npm installed on your computer. Review the [Node.js development environment](#) guide if not.

Then, create a project folder and ensure it contains a `package.json` file, a `build` folder, and an `src` folder. The `src` folder should also include an `index.js` file.

If you have not created a `package.json` file before, navigate to the project folder in the terminal and run the `npm init` command. An interactive prompt will walk you through the steps. Configure as you like, but when it asks for the “entry point”, enter `build/index.js`.

Of course, there are many ways to set up a project using `wp-scripts`, but this is the recommended approach used throughout the Block Editor Handbook.

Finally, install the `wp-scripts` package as a development dependency by running the command:

```
npm install @wordpress/scripts --save-dev
```

Once the installation is complete, your project folder should look like this:

```
example-project-folder/
├── build/
├── node_modules/ (autogenerated)
└── src/
    └── index.js
├── package-lock.json (autogenerated)
└── package.json
```

Basic usage

Once installed, you can run the predefined scripts provided with `wp-scripts` by referencing them in the `scripts` section of your `package.json` file. Here's an example:

```
{
  "scripts": {
    "start": "wp-scripts start",
    "build": "wp-scripts build"
  }
}
```

These scripts can then be run using the command `npm run {script name}`.

The build process with wp-scripts

The two scripts you will use most often are `start` and `build` since they handle the build step. See the [package documentation](#) for all options.

When working on your project, use the `npm run start` command. This will start a development server and automatically rebuild the project whenever any change is detected. Note that the compiled code in `build/index.js` will not be optimized.

When you are ready to deploy your project, use the `npm run build` command. This optimizes your code and makes it production-ready.

After the build finishes, you will see the compiled JavaScript file created at `build/index.js`.

A `build/index.asset.php` file will also be created in the build process, which contains an array of dependencies and a version number (for cache busting). Please, note that to register a block without this `wp-scripts` build process you'll need to manually create `*.asset.php` dependencies files (see [example](#)).

Enqueuing assets

If you register a block via `register_block_type` the scripts defined in `block.json` will be automatically enqueueed (see [example](#))

To manually enqueue files in the editor, in any other context, you can refer to the [Enqueueing assets in the Editor](#) guide for more information, but here's a typical implementation.

```
/***
 * Enqueue Editor assets.
 */
function example_project_enqueue_editor_assets() {
    $asset_file = include( plugin_dir_path( __FILE__ ) . 'build/index.asset.php' );
    wp_enqueue_script(
        'example-editor-scripts',
        plugins_url( 'build/index.js', __FILE__ ),
        $asset_file['dependencies'],
        $asset_file['version']
    );
}
add_action( 'enqueue_block_editor_assets', 'example_project_enqueue_editor_assets' );
```

Here's [an example](#) of manually enqueueing files in the editor.

Next steps

While `start` and `build` will be the two most used scripts, several other useful tools come with `wp-scripts` that are worth exploring. Here's a look at a few.

Maintaining code quality

To help developers improve the quality of their code, `wp-scripts` comes pre-configured with tools like ESLint and Prettier. ESLint ensures your JavaScript adheres to best practices and the

[WordPress coding standards](#), while Prettier automatically formats your code. The available scripts include:

```
{  
  "scripts": {  
    "format": "wp-scripts format",  
    "lint:css": "wp-scripts lint-style",  
    "lint:js": "wp-scripts lint-js",  
  }  
}
```

Regularly linting and formatting your code ensures it's functional, clear, and maintainable for yourself and other developers.

[Running tests](#)

Beyond just writing code, verifying its functionality is crucial. `wp-scripts` includes [Jest](#), a JavaScript testing framework, and both end-to-end and unit testing scripts:

```
{  
  "scripts": {  
    "test:e2e": "wp-scripts test-e2e",  
    "test:unit": "wp-scripts test-unit-js"  
  }  
}
```

Unit tests validate individual units of code, such as functions, ensuring they work as intended, while end-to-end (E2E) tests evaluate the entire project by simulating real-world user scenarios to ensure all parts of the system work seamlessly together.

[Advanced configurations](#)

While `wp-scripts` provides a solid default configuration, there might be cases where you need more specialized setups. The good news is `wp-scripts` is highly adaptable. For example, you can extend and override the default webpack configuration, allowing you to add loaders and plugins or modify almost any part of the build process. This flexibility ensures that as your project grows or its requirements change, `wp-scripts` can be tailored to your evolving needs.

See the `wp-scripts` [package documentation](#) for all configuration options.

[Additional resources](#)

- [@wordpress/scripts](#) (Official documentation)
- [How webpack and WordPress packages interact](#) (WordPress Developer Blog)

First published

October 18, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Get started with wp-scripts”](#)

[Previous Get started with create-block](#) [Previous: Get started with create-block](#)

[Next Quick Start Guide](#) [Next: Quick Start Guide](#)

Quick Start Guide

In this article

[Table of Contents](#)

- [Scaffold the block plugin](#)
- [Basic usage](#)
- [View the block in action](#)
- [Additional resources](#)

[↑ Back to top](#)

This guide is designed to demonstrate the basic principles of block development in WordPress using a hands-on approach. Following the steps below, you will create a custom block plugin that uses modern JavaScript (ESNext and JSX) in a matter of minutes. The example block displays the copyright symbol (©) and the current year, the perfect addition to any website’s footer. You can see these steps in action through this short video demonstration.

[**Scaffold the block plugin**](#)

Start by ensuring you have Node.js and npm installed on your computer. Review the [Node.js development environment](#) guide if not.

Next, use the [@wordpress/create-block](#) package and the [@wordpress/create-block-tutorial-template](#) template to scaffold the complete “Copyright Date Block” plugin.

You can use `create-block` to scaffold a block just about anywhere and then use [wp-env](#) inside the generated plugin folder. This will create a local WordPress development environment with your new block plugin installed and activated.

If you already have your own [local WordPress development environment](#), navigate to the `plugins/` folder using the terminal.

Choose the folder where you want to create the plugin, and then execute the following command in the terminal from within that folder:

```
npx @wordpress/create-block copyright-date-block --template @wordpress/cre
```

The `slug` provided (`copyright-date-block`) defines the folder name for the scaffolded plugin and the internal block name.

Navigate to the Plugins page of your local WordPress installation and activate the “Copyright Date Block” plugin. The example block will then be available in the Editor.

Basic usage

With the plugin activated, you can explore how the block works. Use the following command to move into the newly created plugin folder and start the development process.

```
cd copyright-date-block && npm start
```

When `create-block` scaffolds the block, it installs `wp-scripts` and adds the most common scripts to the block's `package.json` file. Refer to the [Get started with wp-scripts](#) article for an introduction to this package.

The `npm start` command will start a development server and watch for changes in the block's code, rebuilding the block whenever modifications are made.

When you are finished making changes, run the `npm run build` command. This optimizes the block code and makes it production-ready.

View the block in action

You can use any local WordPress development environment to test your new block, but the scaffolded plugin includes configuration for `wp-env`. You must have [Docker](#) already installed and running on your machine, but if you do, run the `npx wp-env start` command.

Once the script finishes running, you can access the local environment at: `http://localhost:8888`. Log into the WordPress dashboard using username `admin` and password `password`. The plugin will already be installed and activated. Open the Editor or Site Editor, and insert the Copyright Date Block as you would any other block.

Visit the [Getting started](#) guide to learn more about `wp-env`.

Additional resources

- [Get started with create-block](#)
- [Get started with wp-scripts](#)
- [Get started with wp-env](#)

First published

November 13, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Quick Start Guide”](#)

[Previous](#) [Get started with wp-scripts](#) [Previous: Get started with wp-scripts](#)
[Next Tutorial: Build your first block](#) [Next: Tutorial: Build your first block](#)

Tutorial: Build your first block

In this article

Table of Contents

- [What you're going to build](#)
- [Prerequisites](#)
- [Scaffolding the block](#)
- [Reviewing the files](#)
- [Initial setup](#)
 - [Updating block.json](#)
 - [Updating index.js](#)
 - [Updating edit.js](#)
 - [Updating render.php](#)
 - [Cleaning up](#)
- [Adding block attributes](#)
 - [Updating block.json](#)
 - [Updating edit.js](#)
 - [Updating render.php](#)
- [Adding static rendering](#)
 - [Why add static rendering?](#)
 - [Adding a save function](#)
 - [Updating save.js](#)
 - [Handling dynamic content in statically rendered blocks](#)
- [Wrapping up](#)

[↑ Back to top](#)

In this tutorial, you will build a “Copyright Date Block”—a basic yet practical block that displays the copyright symbol (©), the current year, and an optional starting year. This type of content is commonly used in website footers.

The tutorial will guide you through the complete process, from scaffolding the block plugin using the [create-block](#) package to modifying each file. While previous WordPress development experience is beneficial, it’s not a prerequisite for this tutorial.

By the end of this guide, you will have a clear understanding of block development fundamentals and the necessary skills to create your own WordPress blocks.

[What you're going to build](#)

Here’s a quick look at what you’re going to build.



Block Tutorial



© 2017–2023

You can also interact with the finished project in [WordPress Playground](#) or use the [Quick Start Guide](#) to install the complete block plugin in your local WordPress environment.

Prerequisites

To complete this tutorial, you will need:

1. Code editor
2. Node.js development tools
3. Local WordPress environment

If you don't have one or more of these items, the [Block Development Environment](#) documentation will help you get started. Come back here once you are all set up.

This tutorial uses [wp-env](#) to create a local WordPress development environment. However, feel free to use alternate local development tools if you already have one that you prefer.

Scaffolding the block

The first step in creating the Copyright Date Block is to scaffold the initial block structure using the [@wordpress/create-block](#) package.

Review the [Get started with create-block](#) documentation for an introduction to using this package.

You can use `create-block` from just about any directory (folder) on your computer and then use `wp-env` to create a local WordPress development environment with your new block plugin installed and activated.

Therefore, choose a directory to place the block plugin or optionally create a new folder called “Block Tutorial”. Open your terminal and `cd` to this directory. Then run the following command.

If you are not using `wp-env`, instead, navigate to the `plugins/` folder in your local WordPress installation using the terminal and run the following command.

```
npx @wordpress/create-block@latest copyright-date-block --variant=dynamic  
cd copyright-date-block
```

After executing this command, you'll find a new directory named `copyright-date-block` in the `plugins` folder. This directory contains all the initial files needed to start customizing your block.

This command also sets up the basic structure of your block, with `copyright-date-block` as its slug. This slug uniquely identifies your block within WordPress.

You might have noticed that the command uses the `--variant=dynamic` flag. This tells `create-block` you want to scaffold a dynamically rendered block. Later in this tutorial, you will learn about dynamic and static rendering and add static rendering to this block.

Navigate to the Plugins page in the WordPress admin and confirm that the plugin is active. Then, create a new page or post and ensure you can insert the Copyright Date Block. It should look like this once inserted.



Block Tutorial



Copyright Date Block – hello from the

Reviewing the files

Before we begin modifying the scaffolded block, it's important to review the plugin's file structure. Open the plugin folder in your code editor.



EXPLORER

...



COPYRIGHT-DATE-BLOCK



> build

> node_modules

src

{ } block.json

JS edit.js

S editor.scss

JS index.js

E render.php

S style.scss

JS view.js

⚙ .editorconfig

NV .gitignore

E copyright-date-block.php

{ } package-lock.json

{ } package.json

i readme.txt

Next, look at the [File structure of a block](#) documentation for a thorough overview of what each file does. Don't worry if this is overwhelming right now. You will learn how to use each file throughout this tutorial.

Since you scaffolded a dynamic block, you will not see a `save.js` file. Later in the tutorial, you will add this file to the plugin to enable static rendering, so stay tuned.

[Initial setup](#)

Let's start by creating the simplest Copyright Date Block possible, which will be a dynamically rendered block that simply displays the copyright symbol (©) and the current year. We'll also add a few controls allowing the user to modify font size and text color.

Before proceeding to the following steps, run `npm run start` in the terminal from within the plugin directory. This command will watch each file in the `/src` folder for changes. The block's build files will be updated each time you save a file.

Check out the [Working with JavaScript for the Block Editor](#) documentation to learn more.

[Updating block.json](#)

Open the `block.json` file in the `/src` folder.

```
{  
  "$schema": "https://schemas.wp.org/trunk/block.json",  
  "apiVersion": 3,  
  "name": "create-block/copyright-date-block",  
  "version": "0.1.0",  
  "title": "Copyright Date Block",  
  "category": "widgets",  
  "icon": "smiley",  
  "description": "Example block scaffolded with Create Block tool.",  
  "example": {},  
  "supports": {  
    "html": false  
  },  
  "textdomain": "copyright-date-block",  
  "editorScript": "file:./index.js",  
  "editorStyle": "file:./index.css",  
  "style": "file:./style-index.css",  
  "render": "file:./render.php",  
  "viewScript": "file:./view.js"  
}
```

Review the [block.json](#) documentation for an introduction to this file.

Since this scaffolding process created this file, it requires some updating to suit the needs of the Copyright Date Block.

Modifying the block identity

Begin by removing the icon and adding a more appropriate description. You will add a custom icon later.

1. Remove the line for icon
2. Update the description to “Display your site’s copyright date.”
3. Save the file

After you refresh the Editor, you should now see that the block no longer has the smiley face icon, and its description has been updated.



Block Tutorial



Copyright Date Block – hello from the

Adding block supports

Next, let's add a few [block supports](#) so that the user can control the font size and text color of the block.

You should always try to use native block supports before building custom functionality. This approach provides users with a consistent editing experience across blocks, and your block benefits from Core functionality with only a few lines of code.

Update the [supports](#) section of the `block.json` file to look like this.

```
"supports": {  
    "color": {  
        "background": false,  
        "text": true  
    },  
    "html": false,  
    "typography": {  
        "fontSize": true  
    }  
},
```

Note that when you enable text color support with `"text": true`, the background color is also enabled by default. You are welcome to keep it enabled, but it's not required for this tutorial, so you can manually set `"background": false`.

Save the file and select the block in the Editor. You will now see both Color and Typography panels in the Settings Sidebar. Try modifying the settings and see what happens.



Block Tutorial



Copyright Date Block – he

Removing unnecessary code

For simplicity, the styling for the Copyright Date Block will be controlled entirely by the color and typography block supports. This block also does not have any front-end Javascript. Therefore, you don't need to specify stylesheets or a `viewScript` in the `block.json` file.

1. Remove the line for `editorStyle`
2. Remove the line for `style`
3. Remove the line for `viewScript`
4. Save the file

Refresh the Editor, and you will see that the block styling now matches your current theme.



Block Tutorial



Copyright Date Block – hello from the

Putting it all together

Your final `block.json` file should look like this:

```
{  
    "$schema": "https://schemas.wp.org/trunk/block.json",  
    "apiVersion": 3,  
    "name": "create-block/copyright-date-block",  
    "version": "0.1.0",  
    "title": "Copyright Date Block",  
    "category": "widgets",  
    "description": "Display your site's copyright date.",  
    "example": {},  
    "supports": {  
        "color": {  
            "background": false,  
            "text": true  
        },  
        "html": false,  
        "typography": {  
            "fontSize": true  
        }  
    },  
    "textdomain": "copyright-date-block",  
    "editorScript": "file:./index.js",  
    "render": "file:./render.php"  
}
```

[Updating index.js](#)

Before you start building the functionality of the block itself, let's do a bit more cleanup and add a custom icon to the block.

Open the [`index.js`](#) file. This is the main JavaScript file of the block and is used to register it on the client. You can learn more about client-side and server-side registration in the [Registration of a block](#) documentation.

Start by looking at the [`registerBlockType`](#) function. This function accepts the name of the block, which we are getting from the imported `block.json` file, and the block configuration object.

```
import Edit from './edit';  
import metadata from './block.json';  
  
registerBlockType( metadata.name, {  
    edit: Edit,  
} );
```

By default, the object just includes the `edit` property, but you can add many more, including `icon`. While most of these properties are already defined in `block.json`, you need to specify the icon here to use a custom SVG.

Adding a custom icon

Using the calendar icon from the [Gutenberg Storybook](#), add the SVG to the function like so:

```
const calendarIcon = (
  <svg
    viewBox="0 0 24 24"
    xmlns="http://www.w3.org/2000/svg"
    aria-hidden="true"
    focusable="false"
  >
    <path d="M19 3H5c-1.1 0-2 .9-2 2v14c0 1.1.9 2 2 2h14c1.1 0 2-.9 2-.9"/>
  </svg>
);

registerBlockType( metadata.name, {
  icon: calendarIcon,
  edit: Edit
} );
```

All block icons should be 24 pixels square. Note the `viewBox` parameter above.

Save the `index.js` file and refresh the Editor. You will now see the calendar icon instead of the default.



Block Tutorial



Copyright Date Block – hello from the

At this point, the block's icon and description are correct, and block supports allow you to change the font size and text color. Now, it's time to move on to the actual functionality of the block.

Updating edit.js

The [edit.js](#) file controls how the block functions and appears in the Editor. Right now, the user sees the message " Copyright Date Block – hello from the editor!". Let's change that.

Open the file and see that the `Edit()` function returns a paragraph tag with the default message.

```
export default function Edit() {
  return (
    <p { ...useBlockProps() }>
      { __(
        'Copyright Date Block - hello from the editor!',
        'copyright-date-block-demo'
      ) }
    </p>
  );
}
```

It looks a bit more complicated than it is.

- [useBlockProps\(\)](#) outputs all the necessary CSS classes and styles in the [block's wrapper](#) needed by the Editor, which includes the style provided by the block supports you added earlier
- [__\(\)](#) is used for the internationalization of text strings

Review the [block wrapper](#) documentation for an introductory guide on how to ensure the block's markup wrapper has the proper attributes.

As a reminder, the main purpose of this block is to display the copyright symbol (©) and the current year. So, you first need to get the current year in string form, which can be done with the following code.

```
const currentYear = new Date().getFullYear().toString();
```

Next, update the function to display the correct information.

```
export default function Edit() {
  const currentYear = new Date().getFullYear().toString();

  return (
    <p { ...useBlockProps() }>© { currentYear }</p>
  );
}
```

Save the `edit.js` file and refresh the Editor. You will now see the copyright symbol (©) followed by the current year.



Block Tutorial



© 2023

Updating render.php

While the block is working properly in the Editor, the default block message is still being displayed on the front end. To fix this, open the `render.php` file, and you will see the following.

```
<?php
...
?>
<p <?php echo get_block_wrapper_attributes(); ?>>© <?php esc_html_e( 'Copyright Date Block - hello from a dynamic block!' );
</p>
```

Similar to the `useBlockProps()` function in the Editor, `get_block_wrapper_attributes()` outputs all the necessary CSS classes and styles in the `block's wrapper`. Only the content needs to be updated.

You can use `date("Y")` to get the current year in PHP, and your `render.php` should look like this.

```
<?php
...
?>
<p <?php echo get_block_wrapper_attributes(); ?>>© <?php echo date( "Y" );
```

Save the file and confirm that the block appears correctly in the Editor and on the front end.

Cleaning up

When you use the `create-block` package to scaffold a block, it might include files that you don't need. In the case of this tutorial, the block doesn't use stylesheets or front end JavaScript. Clean up the plugin's `src/` folder with the following actions.

1. In the `edit.js` file, remove the lines that import `editor.scss`
2. In the `index.js` file, remove the lines that import `style.scss`
3. Delete the `editor.scss`, `style.scss`, and `view.js` files

Finally, make sure that there are no unsaved changes and then terminate the `npm run start` command. Run `npm run build` to optimize your code and make it production-ready.

You have built a fully functional WordPress block, but let's not stop here. In the next sections, we'll add functionality and enable static rendering.

Adding block attributes

The Copyright Date Block you have built shows the current year, but what if you wanted to display a starting year as well?



Block Tutorial



© 2017–2023

This functionality would require users to enter their starting year somewhere on the block. They should also have the ability to toggle it on or off.

You could implement this in different ways, but all would require [block attributes](#). Attributes allow you to store custom data for the block that can then be used to modify the block's markup.

To enable this starting year functionality, you will need one attribute to store the starting year and another that will be used to tell WordPress whether the starting year should be displayed or not.

[Updating block.json](#)

Block attributes are generally specified in the [block.json](#) file. So open up the file and add the following section after the `example` property.

```
"example": {} ,  
"attributes": {  
    "showStartingYear": {  
        "type": "boolean"  
    },  
    "startingYear": {  
        "type": "string"  
    }  
},
```

You must indicate the `type` when defining attributes. In this case, the `showStartingYear` should be true or false, so it's set to `boolean`. The `startingYear` is just a string.

Save the file, and you can now move on to the Editor.

[Updating edit.js](#)

Open the `edit.js` file. You will need to accomplish two tasks.

Add a user interface that allows the user to enter a starting year, toggle the functionality on or off, and store these settings as attributes

Update the block to display the correct content depending on the defined attributes

Adding the user interface

Earlier in this tutorial, you added block supports that automatically created Color and Typography panels in the Settings Sidebar of the block. You can create your own custom panels using the `InspectorControls` component.

Inspector controls

The `InspectorControls` belongs to the [@wordpress/block-editor](#) package, so you can import it into the `edit.js` file by adding the component name on line 14. The result should look like this.

```
import { InspectorControls, useBlockProps } from '@wordpress/block-editor'
```

Next, update the `Edit` function to return the current block content and an `InspectorControls` component that includes the text “Testing.” You can wrap everything in a [`Fragment`](#) (`<></>`) to ensure proper JSX syntax. The result should look like this.

```
export default function Edit() {
  const currentYear = new Date().getFullYear().toString();

  return (
    <>
      <InspectorControls>
        Testing
      </InspectorControls>
      <p { ...useBlockProps() }>© { currentYear }</p>
    </>
  );
}
```

Save the file and refresh the Editor. When selecting the block, you should see the “Testing” message in the Settings Sidebar.



Upd

Page

Block



Copyright Date B

Display your site'



Testing

Advanced

Components and panels

Now, let's use a few more Core components to add a custom panel and the user interface for the starting year functionality. You will want to import [PanelBody](#), [TextControl](#), and [ToggleControl](#) from the [@wordpress/components](#) package.

Add the following line below the other imports in the `edit.js` file.

```
import { PanelBody, TextControl, ToggleControl } from '@wordpress/components'
```

Then wrap the “Testing” message in the `PanelBody` component and set the `title` parameter to “Settings”. Refer to the [component documentation](#) for additional parameter options.

```
export default function Edit() {
    const currentYear = new Date().getFullYear().toString();

    return (
        <>
            <InspectorControls>
                <PanelBody title={ __( 'Settings', 'copyright-date-block' ) }>
                    Testing
                </PanelBody>
            </InspectorControls>
            <p { ...useBlockProps() }>© { currentYear }</p>
        </>
    );
}
```

Save the file and refresh the Editor. You should now see the new Settings panel.



Update

Page

Block



Copyright Date Block

Display your site's copyri



Settings

Testing

Advanced

Text control

The next step is to replace the “Testing” message with a `TextControl` component that allows the user to set the `startingYear` attribute. However, you must include two parameters in the `Edit()` function before doing so.

- `attributes` is an object that contains all the attributes for the block
- `setAttributes` is a function that allows you to update the value of an attribute

With these parameters included, you can fetch the `showStartingYear` and `startingYear` attributes.

Update the top of the `Edit()` function to look like this.

```
export default function Edit( { attributes, setAttributes } ) {  
  const { showStartingYear, startingYear } = attributes;  
  ...
```

To see all the attributes associated with the Copyright Date Block, add `console.log(attributes);` at the top of the `Edit()` function. This can be useful when building and testing a custom block.

Now, you can remove the “Testing” message and add a `TextControl`. It should include:

1. A `label` property set to “Starting year”
2. A `value` property set to the attribute `startingYear`
3. An `onChange` property that updates the `startingYear` attribute whenever the value changes

Putting it all together, the `Edit()` function should look like the following.

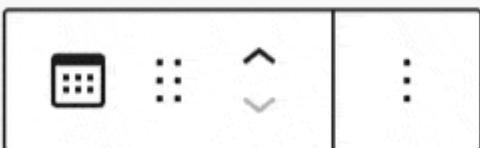
```
export default function Edit( { attributes, setAttributes } ) {  
  const { showStartingYear, startingYear } = attributes;  
  const currentYear = new Date().getFullYear().toString();  
  
  return (  
    <>  
      <InspectorControls>  
        <PanelBody title={ __( 'Settings', 'copyright-date-block' ) }>  
          <TextControl  
            label={ __(  
              'Starting year',  
              'copyright-date-block'  
            ) }  
            value={ startingYear || '' }  
            onChange={ ( value ) =>  
              setAttributes( { startingYear: value } )  
            }  
          />  
        </PanelBody>  
      </InspectorControls>  
      <p { ...useBlockProps() }>© { currentYear }</p>  
    </>  
  );  
}
```

You may have noticed that the `value` property has a value of `startingYear || ''`. The symbol `||` is called the [Logical OR](#) (logical disjunction) operator. This prevents warnings in React when the `startingYear` is empty. See [Controlled and uncontrolled components](#) for details.

Save the file and refresh the Editor. Confirm that a text field now exists in the Settings panel. Add a starting year and confirm that when you update the page, the value is saved.



Block Tutorial



© 2023

Toggle control

Next, let's add a toggle that will turn the starting year functionality on or off. You can do this with a `ToggleControl` component that sets the `showStartingYear` attribute. It should include:

1. A `label` property set to “Show starting year”
2. A `checked` property set to the attribute `showStartingYear`
3. An `onChange` property that updates the `showStartingYear` attribute whenever the toggle is checked or unchecked

You can also update the “Starting year” text input so it's only displayed when `showStartingYear` is `true`, which can be done using the `&&` logical operator.

The `Edit()` function should look like the following.

```
export default function Edit( { attributes, setAttributes } ) {  
  const { showStartingYear, startingYear } = attributes;  
  const currentYear = new Date().getFullYear().toString();  
  
  return (  
    <>  
      <InspectorControls>  
        <PanelBody title={ __( 'Settings', 'copyright-date-block' ) }>  
          <ToggleControl  
            checked={ !! showStartingYear }  
            label={ __(  
              'Show starting year',  
              'copyright-date-block'  
            ) }  
            onChange={ () =>  
              setAttributes( {  
                showStartingYear: ! showStartingYear,  
              } )  
            }  
          />  
          { showStartingYear && (  
            <TextControl  
              label={ __(  
                'Starting year',  
                'copyright-date-block'  
              ) }  
              value={ startingYear || '' }  
              onChange={ ( value ) =>  
                setAttributes( { startingYear: value } )  
              }  
            />  
          ) }  
        </PanelBody>  
      </InspectorControls>  
      <p { ...useBlockProps() }>© { currentYear }</p>  
    </>  
  );  
}
```

Save the file and refresh the Editor. Confirm that clicking the toggle displays the text input, and when you update the page, the toggle remains active.



Block Tutorial



© 2023

Updating the block content

So far, you have created the user interface for adding a starting year and updating the associated block attributes. Now you need to actually update the block content in the Editor.

Let's create a new variable called `displayDate`. When `showStartingYear` is `true` and the user has provided a `startingYear`, the `displayDate` should include the `startingYear` and the `currentYear` separated by an em dash. Otherwise, just display the `currentYear`.

The code should look something like this.

```
let displayDate;

if ( showStartingYear && startingYear ) {
  displayDate = startingYear + '-' + currentYear;
} else {
  displayDate = currentYear;
}
```

When you declare a variable with `let`, it means that the variable may be reassigned later. Declaring a variable with `const` means that the variable will never change. You could rewrite this code using `const`. It's just a matter of personal preference.

Next, you just need to update the block content to use the `displayDate` instead of the `currentYear` variable.

The `Edit()` function should look like the following.

```
export default function Edit( { attributes, setAttributes } ) {
  const { showStartingYear, startingYear } = attributes;
  const currentYear = new Date().getFullYear().toString();

  let displayDate;

  if ( showStartingYear && startingYear ) {
    displayDate = startingYear + '-' + currentYear;
  } else {
    displayDate = currentYear;
  }

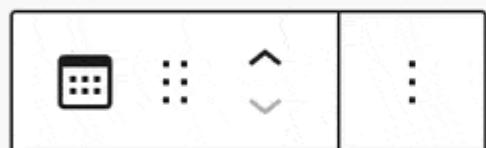
  return (
    <>
      <InspectorControls>
        <PanelBody title={ __( 'Settings', 'copyright-date-block' ) }>
          <ToggleControl
            checked={ !! showStartingYear }
            label={ __(
              'Show starting year',
              'copyright-date-block'
            ) }
            onChange={ () =>
              setAttributes( {
                showStartingYear: ! showStartingYear,
              } )
            }
          />
        </PanelBody>
      </InspectorControls>
    </div>
  );
}
```

```
        } )
    }
  />
{ showStartingYear && (
  <TextControl
    label={ __(
      'Starting year',
      'copyright-date-block'
    ) }
    value={ startingYear || '' }
    onChange={ ( value ) =>
      setAttributes( { startingYear: value } )
    }
  />
) }
</PanelBody>
</InspectorControls>
<p { ...useBlockProps() }>© { displayDate }</p>
</>
);
}
```

Save the file and refresh the Editor. Confirm that the block content updates correctly when you make changes in the Settings panel.



Block Tutorial



© 2019–2023

Updating render.php

While the Editor looks great, the starting year functionality has yet to be added to the front end. Let's fix that by updating the `render.php` file.

Start by adding a variable called `$display_date` and replicate what you did in the `Edit()` function above.

This variable should display the value of the `startingYear` attribute and the `$current_year` variable separated by an em dash, or just the `$current_year` if the `showStartingYear` attribute is `false`.

Three variables are exposed in the `render.php`, which you can use to customize the block's output:

- `$attributes` (array): The block attributes.
- `$content` (string): The block default content.
- `$block` ([WP_Block](#)): The block instance.

The code should look something like this.

```
if ( ! empty( $attributes['startingYear'] ) && ! empty( $attributes['showStartingYear'] ) ) {
    $display_date = $attributes['startingYear'] . '-' . $current_year;
} else {
    $display_date = $current_year;
}
```

Next, you just need to update the block content to use the `$display_date` instead of the `$current_year` variable.

Your final `render.php` file should look like this.

```
<?php
$current_year = date( "Y" );

if ( ! empty( $attributes['startingYear'] ) && ! empty( $attributes['showStartingYear'] ) ) {
    $display_date = $attributes['startingYear'] . '-' . $current_year;
} else {
    $display_date = $current_year;
}
?>
<p <?php echo get_block_wrapper_attributes(); ?>>
    © <?php echo esc_html( $display_date ); ?>
</p>
```

Save the file and confirm that the correct block content is now appearing on the front end of your site.

You have now successfully built a dynamically rendered custom block that utilizes block supports, core WordPress components, and custom attributes. In many situations, this is as far as you would need to go for a block displaying the copyright date with some additional functionality.

In the next section, however, you will add static rendering to the block. This exercise will illustrate how block data is stored in WordPress and provide a fallback should this plugin ever be inadvertently disabled.

Adding static rendering

A block can utilize dynamic rendering, static rendering, or both. The block you have built so far is dynamically rendered. Its block markup and associated attributes are stored in the database, but its HTML output is not.

Statically rendered blocks will always store the block markup, attributes, and output in the database. Blocks can also store static output in the database while being further enhanced dynamically on the front end, a combination of both methods.

You will see the following if you switch to the Code editor from within the Editor.

```
<!-- wp:create-block/copyright-date-block {"showStartingYear":true,"starti
```

Compare this to a statically rendered block like the Paragraph block.

```
<!-- wp:paragraph -->
<p>This is a test.</p>
<!-- /wp:paragraph -->
```

The HTML of the paragraph is stored in post content and saved in the database.

You can learn more about dynamic and static rendering in the [Fundamentals documentation](#). While most blocks are either dynamically or statically rendered, you can build a block that utilizes both methods.

Why add static rendering?

When you add static rendering to a dynamically rendered block, the `render.php` file will still control the output on the front end, but the block's HTML content will be saved in the database. This means that the content will remain if the plugin is ever removed from the site. In the case of this Copyright Date Block, the content will revert to a Custom HTML block that you can easily convert to a Paragraph block.



Block T



Your site doesn't include this block type. To keep it, leave this block intact, or click "Keep as HTML".

Keep as HTML

© 2017–2023

While not necessary in all situations, adding static rendering to a dynamically rendered block can provide a helpful fallback should the plugin ever be disabled unintentionally.

Also, consider a situation where the block markup is included in a block pattern or theme template. If a user installs that theme or uses the pattern without the Copyright Date Block installed, they will get a notice that the block is not available, but the content will still be displayed.

Adding static rendering is also a good exploration of how block content is stored and rendered in WordPress.

[Adding a save function](#)

Start by adding a new file named `save.js` to the `src/` folder. In this file, add the following.

```
import { useBlockProps } from '@wordpress/block-editor';

export default function save() {
    return (
        <p { ...useBlockProps.save() }>
            { 'Copyright Date Block - hello from the saved content!' }
        </p>
    );
}
```

This should look similar to the original `edit.js` file, and you can refer to the [block wrapper](#) documentation for additional information.

Next, in the `index.js` file, import this `save()` function and add a `save` property to the `registerBlockType()` function. Here's a simplified view of the updated file.

```
import save from './save';

...

registerBlockType( metadata.name, {
    icon: calendarIcon,
    edit: Edit,
    save
} );
```

When defining properties of an object, if the property name and the variable name are the same, you can use shorthand property names. This is why the code above uses `save` instead of `save: save`.

Save both `save.js` and `index.js` files and refresh the Editor. It should look like this.



Block Tutorial

:

This block contains unexpected or invalid content.

Don't worry, the error is expected. If you open the inspector in your browser, you should see the following message.

JQMIGRATE: Migrate is installed, version

- ⚠ ► Block validation: Expected ► Object,
- ✖ ► Block validation: Block validation failed (► Object).

Content generated by `save` function:

```
<p class="wp-block-create-block-copyright-content!</p>
```

Content retrieved from post body:

This block validation error occurs because the `save()` function returns block content, but no HTML is stored in the block markup since the previously saved block was dynamic. Remember that this is what the markup currently looks like.

```
<!-- wp:create-block/copyright-date-block {"showStartingYear":true,"starti
```

You will see more of these errors as you update the `save()` function in subsequent steps. Just click “Attempt Block Recovery” and update the page.

After performing block recovery, open the Code editor and you will see the markup now looks like this.

```
<!-- wp:create-block/copyright-date-block {"showStartingYear":true,"starti<p class="wp-block-create-block-copyright-date-block">Copyright Date Block<!-- /wp:create-block/copyright-date-block -->
```

You will often encounter block validation errors when building a block with static rendering, and that's ok. The output of the `save()` function must match the HTML in the post content exactly, which may end up out of sync as you add functionality. So long as there are no validation errors when you're completely finished building the block, you will be all set.

Updating save.js

Next, let's update the output of the `save()` function to display the correct content. Start by copying the same approach used in `edit.js`.

1. Add the `attributes` parameter to the function
2. Define the `showStartingYear` and `startingYear` variables
3. Define a `currentYear` variable
4. Define a `displayDate` variable depending on the values of `currentYear`, `showStartingYear`, and `startingYear`

The result should look like this.

```
export default function save( { attributes } ) {  
  const { showStartingYear, startingYear } = attributes;  
  const currentYear = new Date().getFullYear().toString();  
  
  let displayDate;  
  
  if ( showStartingYear && startingYear ) {  
    displayDate = startingYear + '-' + currentYear;  
  } else {  
    displayDate = currentYear;  
  }  
  
  return (  
    <p { ...useBlockProps.save() }>© { displayDate }</p>  
  );  
}
```

Save the file and refresh the Editor. Click “Attempt Block Recovery” and update the page. Check the Code editor, and the block markup should now look something like this.

```
<!-- wp:create-block/copyright-date-block {"showStartingYear":true,"starti  
<p class="wp-block-create-block-copyright-date-block">© 2017-2023</p>  
<!-- /wp:create-block/copyright-date-block -->
```

At this point, it might look like you're done. The block content is now saved as HTML in the database and the output on the front end is dynamically rendered. However, there are still a few things that need to be addressed.

Consider the situation where the user added the block to a page in 2023 and then went back to edit the page in 2024. The front end will update as expected, but in the Editor, there will be a block validation error. The `save()` function knows that it's 2024, but the block content saved in the database still says 2023.

Let's fix this in the next section.

Handling dynamic content in statically rendered blocks

Generally, you want to avoid dynamic content in statically rendered blocks. This is part of the reason why the term “dynamic” is used when referring to dynamic rendering.

That said, in this tutorial, you are combining both rendering methods, and you just need a bit more code to avoid any block validation errors when the year changes.

The root of the issue is that the `currentYear` variable is set dynamically in the `save()` function. Instead, this should be a static variable within the function, which can be solved with an additional attribute.

Adding a new attribute

Open the `block.json` file and add a new attribute called `fallbackCurrentYear` with the type `string`. The `attributes` section of the file should now look like this.

```
"attributes": {  
    "fallbackCurrentYear": {  
        "type": "string"  
    },  
    "showStartingYear": {  
        "type": "boolean"  
    },  
    "startingYear": {  
        "type": "string"  
    }  
},
```

Next, open the `save.js` file and use the new `fallbackCurrentYear` attribute in place of `currentYear`. Your updated `save()` function should look like this.

```
export default function save( { attributes } ) {  
    const { fallbackCurrentYear, showStartingYear, startingYear } = attributes;  
  
    let displayDate;  
  
    if ( showStartingYear && startingYear ) {  
        displayDate = startingYear + '-' + fallbackCurrentYear;  
    } else {  
        displayDate = fallbackCurrentYear;  
    }  
  
    return (  
        <p { ...useBlockProps.save() }>© { displayDate }</p>  
    );  
}
```

Now, what happens if the `fallbackCurrentYear` is undefined?

Before the `currentYear` was defined within the function, so the `save()` function always had content to return, even if `showStartingYear` and `startingYear` were undefined.

Instead of returning just the copyright symbol, let's add a condition that if `fallbackCurrentYear` is not set, return `null`. It's generally better to save no HTML in the database than incomplete data.

The final `save()` function should look like this.

```

export default function save( { attributes } ) {
  const { fallbackCurrentYear, showStartingYear, startingYear } = attributes;

  if ( ! fallbackCurrentYear ) {
    return null;
  }

  let displayDate;

  if ( showStartingYear && startingYear ) {
    displayDate = startingYear + '-' + fallbackCurrentYear;
  } else {
    displayDate = fallbackCurrentYear;
  }

  return (
    <p { ...useBlockProps.save() }>© { displayDate }</p>
  );
}

```

Save both the `block.json` and `save.js` files; you won't need to make any more changes.

Setting the attribute in `edit.js`

The `save()` function now uses the new `fallbackCurrentYear`, so it needs to be set somewhere. Let's use the `Edit()` function.

Open the `edit.js` file and start by defining the `fallbackCurrentYear` variable at the top of the `Edit()` functional alongside the other attributes. Next, review what's happening in the function.

When the block loads in the Editor, the `currentYear` variable is defined. The function then uses this variable to set the content of the block.

Now, let's set the `fallbackCurrentYear` attribute to the `currentYear` when the block loads if the attribute is not already set.

```

if ( currentYear !== fallbackCurrentYear ) {
  setAttributes( { fallbackCurrentYear: currentYear } );
}

```

This will work but can be improved by ensuring this code only runs once when the block is initialized. To do so, you can use the [useEffect](#) React hook. Refer to the React documentation for more information about how to use this hook.

First, import `useEffect` with the following code.

```
import { useEffect } from 'react';
```

Then wrap the `setAttribute()` code above in a `useEffect` and place this code after the `currentYear` definition in the `Edit()` function. The result should look like this.

```

export default function Edit( { attributes, setAttributes } ) {
  const { fallbackCurrentYear, showStartingYear, startingYear } = attributes;

```

```

// Get the current year and make sure it's a string.
const currentYear = new Date().getFullYear().toString();

// When the block loads, set the fallbackCurrentYear attribute to the
// current year if it's not already set.
useEffect( () => {
    if ( currentYear !== fallbackCurrentYear ) {
        setAttributes( { fallbackCurrentYear: currentYear } );
    }
}, [ currentYear, fallbackCurrentYear, setAttributes ] );

...

```

When the block is initialized in the Editor, the `fallbackCurrentYear` attribute will be immediately set. This value will then be available to the `save()` function, and the correct block content will be displayed without block validation errors.

The one caveat is when the year changes. If a Copyright Date Block was added to a page in 2023 and then edited in 2024, the `fallbackCurrentYear` attribute will no longer equal the `currentYear`, and the attribute will be automatically updated to 2024. This will update the HTML returned by the `save()` function.

You will not get any block validation errors, but the Editor will detect that changes have been made to the page and prompt you to update.

Optimizing render.php

The final step is to optimize the `render.php` file. If the `currentYear` and the `fallbackCurrentYear` attribute are the same, then there is no need to dynamically create the block content. It is already saved in the database and is available in the `render.php` file via the `$content` variable.

Therefore, update the file to render the generated content if `currentYear` and `fallbackCurrentYear` do not match.

```

$current_year = date( "Y" );

// Determine which content to display.
if ( isset( $attributes['fallbackCurrentYear'] ) && $attributes['fallbackC
    // The current year is the same as the fallback, so use the block cont
    $block_content = $content;
} else {

    // The current year is different from the fallback, so render the upda
    if ( ! empty( $attributes['startingYear'] ) && ! empty( $attributes['s
        $display_date = $attributes['startingYear'] . '-' . $current_year;
    } else {
        $display_date = $current_year;
    }

    $block_content = '<p ' . get_block_wrapper_attributes() . '>© ' . esc_
}

```

```
echo wp_kses_post( $block_content );
```

That's it! You now have a block that utilizes both dynamic and static rendering.

Wrapping up

Congratulations on completing this tutorial and building your very own Copyright Date Block. Throughout this journey, you have gained a solid foundation in WordPress block development and are now ready to start building your own blocks.

For final reference, the complete code for this tutorial is available in the [Block Development Examples](#) repository on GitHub.

Now, whether you're now looking to refine your skills, tackle more advanced projects, or stay updated with the latest WordPress trends, the following resources will help you improve your block development skills:

- [Block Development Environment](#)
- [Fundamentals of Block Development](#)
- [WordPress Developer Blog](#)
- [Block Development Examples](#) | GitHub repository

Remember, every expert was once a beginner. Keep learning, experimenting, and, most importantly, have fun building with WordPress.

First published

December 15, 2023

Last updated

January 28, 2024

Edit article

[Improve it on GitHub: Tutorial: Build your first block”](#)

[Previous Quick Start Guide](#) [Previous: Quick Start Guide](#)

[Next Fundamentals of Block Development](#) [Next: Fundamentals of Block Development](#)

Fundamentals of Block Development

[↑ Back to top](#)

This section provides an introduction to the most relevant concepts in Block Development.

In this section, you will learn:

1. **[File structure of a block](#)** – The purpose of each one of the types of files available for a block, the relationships between them, and their role in the output of the block.

2. [**block.json**](#) – How a block is defined using its `block.json` metadata and some relevant properties of this file (such as `attributes` and `supports`).
3. [**Registration of a block**](#) – How a block is registered in both the server and the client.
4. [**Block wrapper**](#) – How to set proper attributes to the block's markup wrapper.
5. [**The block in the Editor**](#) – The block as a React component loaded in the Block Editor and its possibilities.
6. [**Markup representation of a block**](#) – How blocks are represented in the database, theme templates, or patterns.
7. [**Static or Dynamic rendering of a block**](#) – How blocks can generate their output for the front end dynamically or statically.
8. [**Javascript in the Block Editor**](#) – How to work with Javascript for the Block Editor.

First published

November 28, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Fundamentals of Block Development”](#)

[Previous Tutorial: Build your first block](#) [Previous: Tutorial: Build your first block](#)

[Next File structure of a block](#) [Next: File structure of a block](#)

File structure of a block

In this article

Table of Contents

- [**Folders and files involved in a block's definition and registration**](#)
 - [`<plugin-file>.php`](#)
 - [`package.json`](#)
 - [`src folder`](#)
 - [`block.json`](#)
 - [`index.js`](#)
 - [`edit.js`](#)
 - [`save.js`](#)
 - [`style.\(css|scss|sass\)`](#)
 - [`editor.\(css|scss|sass\)`](#)
 - [`render.php`](#)
 - [`view.js`](#)
 - [`build folder`](#)
- [**Additional resources**](#)

[↑ Back to top](#)

It is recommended to **register blocks within plugins** to ensure they stay available when a theme gets switched. With the [create-block tool](#) you can quickly scaffold the structure of the files required to create a plugin that registers a block.

The files generated by `create-block` are a good reference of the files that can be involved in the definition and registration of a block.

File Structure of

At its simplest, a block in the WordPress bl

Folders and files involved in a block's definition and registration

<plugin-file>.php

A block is usually added to the block editor using a WordPress plugin. In the main PHP file of the plugin the block is usually registered on the server side.

For more on creating a WordPress plugin see [Plugin Basics](#), and [Plugin Header requirements](#) for explanation and additional fields you can include in your plugin header.

package.json

[package.json](#) is a configuration file for a Node.js project. In this file you define the NPM dependencies of the block and the scripts used for local work.

src folder

In a standard project you'll place your block files in the `src` folder. By default, [the build process with wp-scripts](#) will take files from this folder and will generate the bundled files in the `build` folder.

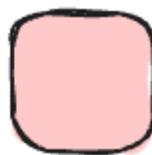
block.json

This file contains the [metadata of the block](#), and it's used to simplify the definition and registration of the block both in the client and on the server.

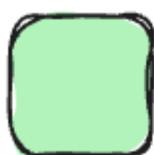
Among other data it provides properties to define the paths of the files involved in the block's behaviour, output and style. If there's a build process involved, this `block.json` along with the generated files are placed into a destination folder (usually the `build` folder) so the paths provided target to the bundled versions of these files.

The most relevant properties that can be defined in a `block.json` to set the files involved in the block's behaviour, output, or style are:

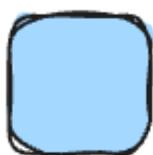
- The [editorScript](#) property, usually set with the path of a bundled `index.js` file (output build from `src/index.js`).
- The [style](#) property, usually set with the path of a bundled `style-index.css` file (output build from `src/style.(css|scss|sass)`).
- The [editorStyle](#) property, usually set with the path of a bundled `index.css` (output build from `src/editor.(css|scss|sass)`).
- The [render](#) property, usually set with the path of a bundled `render.php` (output copied from `src/render.php`).
- The [viewScript](#) property, usually set with the path of a bundled `view.js` (output copied from `src/view.php`).



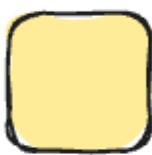
plugin file



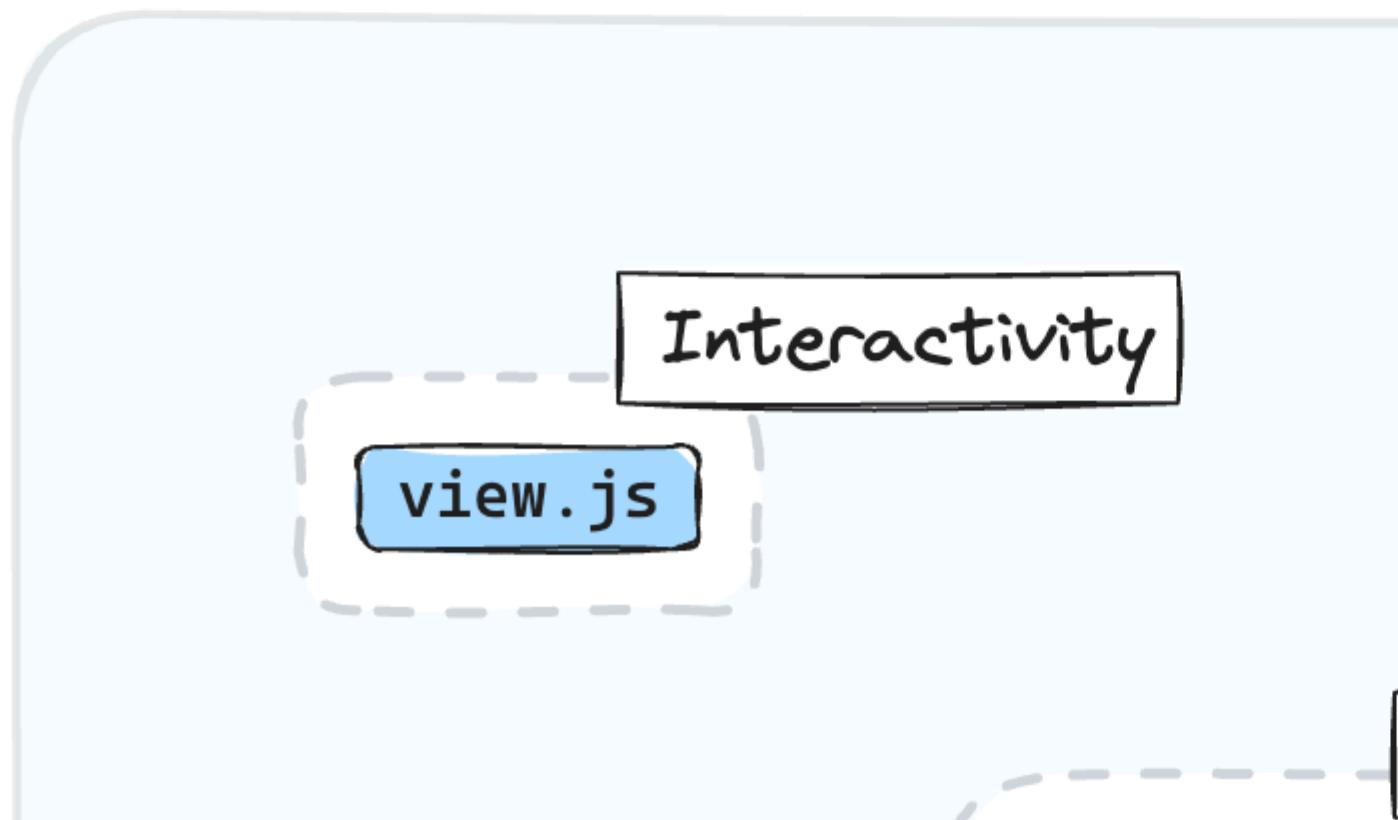
package file



block source



block genera



[index.js](#)

The `index.js` file (or any other file defined in the `editorScript` property of `block.json`) is the entry point file for javascript that should only get loaded in the editor. It is responsible for calling the `registerBlockType` function to register the block on the client. In a standard structure it imports the `edit.js` and `save.js` files to get functions required in block registration.

[edit.js](#)

The `edit.js` commonly gets used to contain the React component that gets used in the editor for our block. It usually exports a single component that then gets passed to the [edit property of the registerBlockType](#) function in the `index.js` file.

[save.js](#)

The `save.js` exports the function that returns the static HTML markup that gets saved to the Database and that is passed to the [save property of the registerBlockType](#) function in the `index.js` file.

[style.\(css|scss|sass\)](#)

A `style` file with any of the extensions `.css`, `.scss` or `.sass`, contains the styles of the block that will be loaded in both the editor and the frontend. In the build process this file is converted into `style-index.css` which is usually defined at [style](#) property in `block.json`

The webpack config used internally by `wp-scripts` includes a [css-loader](#) chained with `postcss-loader` and [sass-loader](#) that allows it to process CSS, SASS or SCSS files. Check [Default webpack config](#) for more info

[editor.\(css|scss|sass\)](#)

An `editor` file with any of the extensions `.css`, `.scss` or `.sass`, contains the additional styles applied to the block only in the editor's context. In the build process this file is converted into `index.css` which is usually defined at [editorStyle](#) property in `block.json`

[render.php](#)

The `render.php` file (or any other file defined in the [render](#) property of `block.json`) defines the server side process that returns the markup for the block when there is a request from the frontend. If this file is defined, it will take precedence over any other ways to render the block's markup for the frontend.

[view.js](#)

The `view.js` file (or any other file defined in the [viewScript](#) property of `block.json`) will be loaded in the front-end when the block is displayed.

[build folder](#)

In a standard project, the `build` folder contains the generated files in [the build process triggered by the build or start commands of wp-scripts](#).

You can use `webpack-src-dir` and `output-path` option of `wp-scripts` build commands to [customize the entry and output points](#)

[Additional resources](#)

- [File Structure of a Block diagram](#)

First published

November 28, 2023

Last updated

January 25, 2024

Edit article

[Improve it on GitHub: File structure of a block”](#)

[Previous Fundamentals of Block Development](#) [Previous: Fundamentals of Block Development](#)
[Next block.json](#) [Next: block.json](#)

block.json

In this article

[Table of Contents](#)

- [Basic metadata of the block](#)
- [Files for the block’s behavior, output, or style](#)
- [Using attributes to store block data](#)
 - [Reading and updating attributes](#)
- [Enable UI settings panels for the block with supports](#)
- [Additional resources](#)

[↑ Back to top](#)

The `block.json` file simplifies the processs of defining and registering a block by using the same block’s definition in JSON format to register the block in both the server and the client.

The block.json file

→ contains block metadata used to register the block

attributes object to declare data stored by your block
Attributes are declared in the form of key/value pairs

Key
(name of the attribute)

```
attributes: {  
    url: {  
        type: 'string',  
        source: 'attribute',  
        selector: 'img',  
        attribute: 'src',  
    },  
    title: {  
        type: 'string',  
    },  
    size: {  
        enum: [ 'large', 'small' ],  
    },  
}
```

set of options to control features used in the editor
supports object to declare support for certain features

```
supports: {  
    color: true  
}
```

Click [here](#) to see a full block example and check [its block.json](#)

Besides simplifying a block's registration, using a `block.json` has [several benefits](#), including improved performance and development.

At [Metadata in block.json](#) you can find a detailed explanation of all the properties you can set in a `block.json` for a block. With these properties you can define things such as:

- Basic metadata of the block
- Files for the block's behavior, style, or output
- Data Storage in the Block
- Setting UI panels for the block

[Basic metadata of the block](#)

Through properties of the `block.json`, we can define how the block will be uniquely identified, how it can be found, and the info displayed for the block in the Block Editor. Some of these properties are:

- [apiVersion](#): the version of [the API](#) used by the block (current version is 2).
- [name](#): a unique identifier for a block, including a namespace.
- [title](#): a display title for a block.
- [category](#): a block category for the block in the Inserter panel.
- [icon](#): a [Dashicon](#) slug or a custom SVG icon.
- [description](#): a short description visible in the block inspector.
- [keywords](#): to locate the block in the inserter.
- [textdomain](#): the plugin text-domain (important for things such as translations).

[Files for the block's behavior, output, or style](#)

The [editorScript](#) and [editorStyle](#) properties allow defining Javascript and CSS files to be enqueued and loaded **only in the editor**.

The [script](#) and [style](#) properties allow the definition of Javascript and CSS files to be enqueued and loaded **in both the editor and the front end**.

The [viewScript](#) property allow us to define the Javascript file or files to be enqueued and loaded **only in the front end**.

All these properties (`editorScript`, `editorStyle`, `script` `style`, `viewScript`) accept as a value a [path for the file](#) (prefixed with `file:`), a [handle registered with wp_register_script or wp_register_style](#), or an array with a mix of both.

The [render](#) property ([introduced on WordPress 6.1](#)) sets the path of a `.php` template file that will render the markup returned to the front end. This only method will be used to return the markup for the block on request only if `$render_callback` function has not been passed to the `register_block_type` function.

[Using attributes to store block data](#)

Block [attributes](#) are settings or data assigned to blocks. They can determine various aspects of a block, such as its content, layout, style, and any other specific information you need to store along with your block's structure. If the user changes a block, such as modifying the font size, you need a way to persist these changes. Attributes are the solution.

When registering a new block type, the `attributes` property of `block.json` describes the custom data the block requires and how they're stored in the database. This allows the Editor to parse these values correctly and pass the `attributes` to the block's `Edit` and `save` functions.

Example: Attributes as defined in block.json

```
"attributes": {  
    "fallbackCurrentYear": {  
        "type": "string"  
    },  
    "showStartingYear": {  
        "type": "boolean"  
    },  
    "startingYear": {  
        "type": "string"  
    }  
},
```

By default, attributes are serialized and stored in the block's delimiter, but this [can be configured](#).

Example: Atributes stored in the Markup representation of the block

```
<!-- wp:block-development-examples/copyright-date-block-09aac3 {"fallbackC  
<p class="wp-block-block-development-examples-copyright-date-block-09aac3"  
<!-- /wp:block-development-examples/copyright-date-block-09aac3 -->x
```

[Reading and updating attributes](#)

These [attributes](#) are passed to the React component `Edit` (to display in the Block Editor) and the `save` function (to return the markup saved to the database) of the block, and to any server-side render definition for the block (see the `render` property above).

The `Edit` component receives exclusively the capability of updating the attributes via the [setAttributes](#) function.

See how the attributes are passed to the [Edit component](#), [the save function](#) and [the render.php](#) in this [full block example](#) of the code above

Check the [attributes](#) reference page for full info about the Attributes API.

```
"attributes":  
  "content": {  
    "type": "st  
  "source": "  
  "selector":
```

```
}
```

```
,
```

Block Editor

edit.js

```
Edit = ( { attributes, setA
```

```
onChange = { (newContent) => setA
```

Enable UI settings panels for the block with supports

Many blocks, including core blocks, offer similar customization options, whether changing the background color, text color, or adding padding customization options.

The [supports](#) property in `block.json` allows a block to declare support for certain features, enabling users to customize specific settings (like colors or margins) from the Settings Sidebar.

Using the available block `supports` allows you to align your block's behavior with core blocks and avoid replicating the same functionality yourself.

Example: Supports as defined in block.json

```
"supports": {  
    "color": {  
        "text": true,  
        "link": true,  
        "background": true  
    }  
}
```

The use of `supports` generates a set of properties that need to be manually added to the [wrapping element of the block](#). This ensures they're properly stored as part of the block data and taken into account when generating the markup of the block that will be delivered to the front end.

Example: Supports custom settings stored in the Markup representation of the block

```
<!-- wp:block-development-examples/block-supports-6aa4dd {"backgroundColor": "#f0f0f0"} -->

<!-- /wp:block-development-examples/block-supports-6aa4dd --&gt;</p>


```

See the [full block example](#) of the [code above](#)

Check the [supports](#) reference page for full info about the Supports API.

Additional resources

- [block.json diagram](#)
- [Attributes diagram](#)

First published

November 29, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: block.json](#)

[Previous File structure of a block](#) [Previous: File structure of a block](#)

Registration of a block

In this article

Table of Contents

- [Registration of the block with PHP \(server-side\)](#)
- [Registration of the block with JavaScript \(client-side\)](#)
- [Additional resources](#)

[↑ Back to top](#)

A block is usually registered through a plugin on both the server and client-side using its `block.json` metadata.

Although technically, blocks could be registered only in the client, **registering blocks on both the server and in the client is a strong recommendation**. Some server-side features like Dynamic Rendering, Block Supports, Block Hooks, or Block style variations require the block to “exist” on the server, and they won’t work properly without server registration of the block.

For example, to allow a block [to be styled via `theme.json`](#), it needs to be registered on the server, otherwise, any styles assigned to it in `theme.json` will be ignored.

From the plugin file

path to the folder

Registration of the block with PHP (server-side)

Block registration on the server usually takes place in the main plugin PHP file with the `register_block_type` function called on the [init hook](#).

The `register_block_type` function aims to simplify block type registration on the server by reading metadata stored in the `block.json` file.

This function takes two params relevant in this context (`$block_type` accepts more types and variants):

- `$block_type (string)` – path to the folder where the `block.json` file is located or full path to the metadata file if named differently.
- `$args (array)` – an optional array of block type arguments. Default value: `[]`. Any arguments may be defined. However, the one described below is supported by default:
 - `$render_callback (callable)` – callback used to render blocks of this block type, it's an alternative to the `render` field in `block.json`.

As part of the build process, the `block.json` file is usually copied from the `src` folder to the `build` folder, so the path to the `block.json` of your registered block should refer to the `build` folder.

`register_block_type` returns the registered block type (`WP_Block_Type`) on success or `false` on failure.

Example:

```
register_block_type(  
    __DIR__ . '/notice',  
    array(  
        'render_callback' => 'render_block_core_notice',  
    )  
) ;
```

Example:

```
function minimal_block_ca6eda_register_block() {  
    register_block_type( __DIR__ . '/build' );  
}  
  
add_action( 'init', 'minimal_block_ca6eda_register_block' );
```

See the [full block example](#) of the [code above](#)

Registration of the block with JavaScript (client-side)

When the block is registered on the server, you only need to register the client-side settings on the client using the same block's name.

Example:

```
registerBlockType( 'my-plugin/notice', {  
    edit: Edit,
```

```
// ...other client-side settings  
} );
```

Although registering the block also on the server with PHP is still recommended for the reasons mentioned at [“Benefits using the metadata file”](#), if you want to register it only client-side you can use [registerBlockType](#) method from `@wordpress/blocks` package to register a block type using the metadata loaded from `block.json` file.

The function takes two params:

- `$blockNameOrMetadata(string|Object)` – block type name or the metadata object loaded from the `block.json`
- `$settings (Object)` – client-side block settings.

The content of `block.json` (or any other `.json` file) can be imported directly into Javascript files when using [a build process like the one available with wp-scripts](#)

The client-side block settings object passed as a second parameter includes two especially relevant properties:

- `edit`: The React component that gets used in the editor for our block.
- `save`: The function that returns the static HTML markup that gets saved to the Database.

`registerBlockType` returns the registered block type (`WPBlock`) on success or `undefined` on failure.

Example:

```
import { registerBlockType } from '@wordpress/blocks';  
import { useBlockProps } from '@wordpress/block-editor';  
import metadata from './block.json';  
  
const Edit = () => <p { ...useBlockProps() }>Hello World - Block Editor</p>  
const save = () => <p { ...useBlockProps.save() }>Hello World - Frontend</p>  
  
registerBlockType( metadata.name, {  
    edit: Edit,  
    save,  
} );
```

See the [code above](#) in an example

Additional resources

- [register_block_type PHP function](#)
- [registerBlockType JS function](#)
- [Why a block needs to be registered in both the server and the client?](#) | GitHub Discussion
- [Block Registration diagram](#)

First published

November 28, 2023

Last updated

January 25, 2024

[Edit article](#)

[Improve it on GitHub: Registration of a block”](#)

[Previous block.json](#) [Previous: block.json](#)

[Next The block wrapper](#) [Next: The block wrapper](#)

The block wrapper

In this article

[Table of Contents](#)

- [The Edit component’s markup](#)
- [The Save component’s markup](#)
- [The server-side render markup](#)

[↑ Back to top](#)

Each block’s markup is wrapped by a container HTML tag that needs to have the proper attributes to fully work in the Block Editor and to reflect the proper block’s style settings when rendered in the Block Editor and the front end. As developers, we have full control over the block’s markup, and WordPress provides the tools to add the attributes that need to exist on the wrapper to our block’s markup.

Ensuring proper attributes to the block wrapper is especially important when using custom styling or features like `supports`.

The use of `supports` generates a set of properties that need to be manually added to the wrapping element of the block so they’re properly stored as part of the block data.

A block can have three sets of markup defined, each one of them with a specific target and purpose:

- The one for the **Block Editor**, defined through a `edit` React component passed to `registerBlockType` when registering the block in the client.
- The one used to **save the block in the DB**, defined through a `save` function passed to `registerBlockType` when registering the block in the client.
 - This markup will be returned to the front end on request if no dynamic render has been defined for the block.
- The one used to **dynamically render the markup of the block** returned to the front end on request, defined through the `render_callback` on `register_block_type` or the `render` PHP file in `block.json`
 - If defined, this server-side generated markup will be returned to the front end, ignoring the markup stored in DB.

For the `edit React component` and the `save function`, the block wrapper element should be a native DOM element (like `<div>`) or a React component that forwards any additional props to native DOM elements. Using a `<Fragment>` or `<ServerSideRender>` component, for instance, would be invalid.

The Edit component's markup

The `useBlockProps()` hook available on the [@wordpress/block-editor](#) allows passing the required attributes for the Block Editor to the `edit` block's outer wrapper.

Among other things, the `useBlockProps()` hook takes care of including in this wrapper:

- An `id` for the block's markup
- Some accessibility and `data-` attributes
- Classes and inline styles reflecting custom settings, which include by default:
 - The `wp-block` class
 - A class that contains the name of the block with its namespace

For example, for the following piece of code of a block's registration in the client...

```
const Edit = () => <p { ...useBlockProps() }>Hello World - Block Editor</p>

registerBlockType( ..., {
  edit: Edit
} );
```

(see the [code above in an example](#))

...the markup of the block in the Block Editor could look like this:

```
<p
  tabindex="0"
  id="block-4462939a-b918-44bb-9b7c-35a0db5ab8fe"
  role="document"
  aria-label="Block: Minimal Gutenberg Block ca6eda"
  data-block="4462939a-b918-44bb-9b7c-35a0db5ab8fe"
  data-type="block-development-examples/minimal-block-ca6eda"
  data-title="Minimal Gutenberg Block ca6eda"
  class=""
    block-editor-block-list__block
    wp-block
    is-selected
    wp-block-block-development-examples-minimal-block-ca6eda
  "
>Hello World - Block Editor</p>
```

Any additional classes and attributes for the `Edit` component of the block should be passed as an argument of `useBlockProps` (see [example](#)). When you add `supports` for any feature, they get added to the object returned by the `useBlockProps` hook.

The Save component's markup

When saving the markup in the DB, it's important to add the block props returned by `useBlockProps.save()` to the wrapper element of your block.
`useBlockProps.save()` ensures that the block class name is rendered properly in addition to any HTML attribute injected by the block supports API.

For example, for the following piece of code of a block's registration in the client that defines the markup desired for the DB (and returned to the front end by default)...

```
const Edit = () => <p { ...useBlockProps() }>Hello World - Block Editor</p>
const save = () => <p { ...useBlockProps.save() }>Hello World - Frontend</p>

registerBlockType( ..., {
  edit,
  save,
} );
```

(see the [code above in an example](#))

...the markup of the block in the front end could look like this:

```
<p class="wp-block-block-development-examples-minimal-block-ca6eda">Hello
```

Any additional classes and attributes for the `save` function of the block should be passed as an argument of `useBlockProps.save()` (see [example](#)).

When you add `supports` for any feature, the proper classes get added to the object returned by the `useBlockProps.save()` hook.

```
<p class="wp-block-block-development-examples-block-supports-6aa4dd has-accent-4-color has-contrast-background-color has-text-color has-background">Hello World</p>
```

(check the [example](#) that generated the HTML above in the front end)

The server-side render markup

Any markup in the server-side render definition for the block can use the `get_block_wrapper_attributes()` function to generate the string of attributes required to reflect the block settings (see [example](#)).

```
<p <?php echo get_block_wrapper_attributes(); ?>>
  <?php esc_html_e( 'Block with Dynamic Rendering - hello!!!', 'block-de
```

First published

November 29, 2023

Last updated

January 25, 2024

Edit article

[Improve it on GitHub: The block wrapper](#)"

The block in the Editor

In this article

Table of Contents

- [Built-in components](#)
- [Block Controls: Block Toolbar and Settings Sidebar](#)
 - [Block Toolbar](#)
 - [Settings Sidebar](#)
- [Additional resources](#)

[↑ Back to top](#)

The Block Editor is a React Single Page Application (SPA) and every block in the editor is displayed through a React component defined in the `edit` property of the settings object used to [register the block on the client](#).

The `props` object received by the block's `Edit` React component includes:

- [attributes](#) – attributes object
- [setAttributes](#) – method to update the attributes object
- [isSelected](#) – boolean that communicates whether the block is currently selected

WordPress provides many built-in standard components that can be used to define the interface of the block in the editor. These built-in components are available via packages such as [@wordpress/components](#) and [@wordpress/block-editor](#).

The WordPress Gutenberg project uses [Storybook](#) to document the user interface components that are available in WordPress packages.

Custom settings controls for the block in the Block Toolbar or the Settings Sidebar can also be defined through this `Edit` React component via built-in components such as:

- [InspectorControls](#)
- [BlockControls](#)

[Built-in components](#)

The package [@wordpress/components](#) includes a library of generic WordPress components to create common UI elements for the Block Editor and the WordPress dashboard. Some of the most commonly used components from this package are:

- [TextControl](#)
- [Panel](#)
- [ToggleControl](#)
- [ExternalLink](#)

The package [@wordpress/block-editor](#) includes a library of components and hooks for the Block Editor, including those to define custom settings controls for the block in the Editor. Some of the components most commonly used from this package are:

- [RichText](#)
- [BlockControls](#)
- [InspectorControls](#)
- [InnerBlocks](#)
- PanelColorSettings or ColorPalette

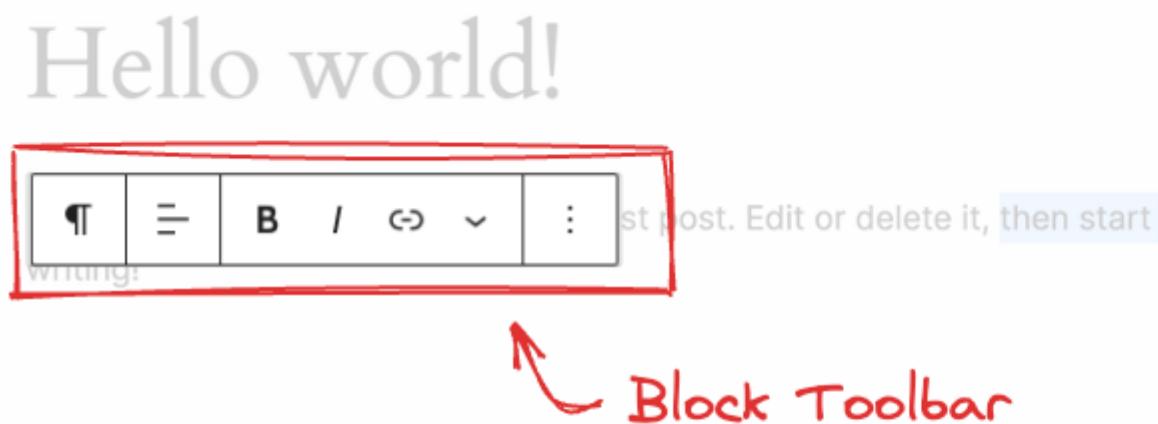
The package [@wordpress/block-editor](#) also provides the tools to create and use standalone block editors.

A good workflow when using a component for the Block Editor is:

- Import the component from a WordPress package
- Add the corresponding code for the component to your project in JSX format
- Most built-in components will be used to set [block attributes](#), so define any necessary attributes in `block.json` and create event handlers to update those attributes with `setAttributes` in your component
- If needed, adapt the code to be serialized and stored in the database

[Block Controls: Block Toolbar and Settings Sidebar](#)

To simplify block customization and ensure a consistent experience for users, there are a number of built-in UI patterns to help generate the editor preview.



Block Toolbar

When the user selects a block, a number of control buttons may be shown in a toolbar above the selected block. Some of these block-level controls may be included automatically but you can also customize the toolbar to include controls specific to your block type. If the return value of your block type's `edit` function includes a `BlockControls` element, those controls will be shown in the selected block's toolbar.

```
export default function Edit( { className, attributes: attr, setAttributes } ) {  
  const onChangeContent = ( newContent ) => {  
    setAttributes( { content: newContent } );  
  };  
  
  const onChangeAlignment = ( newAlignment ) => {  
    setAttributes( {  
      alignment: newAlignment === undefined ? 'none' : newAlignment,  
    } );  
  };  
  
  return (  
    <div { ...useBlockProps() }>  
      <BlockControls>  
        <ToolbarGroup>  
          <AlignmentToolbar  
            value={ attr.alignment }  
            onChange={ onChangeAlignment }  
          />  
        </ToolbarGroup>  
      </BlockControls>  
  
      <RichText  
        className={ className }  
        style={ { textAlign: attr.alignment } }  
        tagName="p"  
        onChange={ onChangeContent }  
        value={ attr.content }  
      />  
    </div>  
  );  
}
```

See the [full block example](#) of the code above.

Note that `BlockControls` is only visible when the block is currently selected and in visual editing mode. `BlockControls` are not shown when editing a block in HTML editing mode.

Settings Sidebar

The Settings Sidebar is used to display less-often-used settings or settings that require more screen space. The Settings Sidebar should be used for **block-level settings only**.

If you have settings that affects only selected content inside a block (example: the “bold” setting for selected text inside a paragraph): **do not place it inside the Settings Sidebar**. The Settings

Sidebar is displayed even when editing a block in HTML mode, so it should only contain block-level settings.

The Block Tab is shown in place of the Document Tab when a block is selected.

Similar to rendering a toolbar, if you include an `InspectorControls` element in the return value of your block type's `edit` function, those controls will be shown in the Settings Sidebar region.

```
export default function Edit( { attributes, setAttributes } ) {
    const onChangeBGCOLOR = ( hexColor ) => {
        setAttributes( { bg_color: hexColor } );
    };

    const onChangeTextColor = ( hexColor ) => {
        setAttributes( { text_color: hexColor } );
    };

    return (
        <div { ...useBlockProps() }>
            <InspectorControls key="setting">
                <div>
                    <fieldset>
                        <legend className="blocks-base-control__label">
                            { __( 'Background color', 'block-development-example' ) }
                        </legend>
                        <ColorPalette // Element Tag for Gutenberg standard
                            onChange={ onChangeBGCOLOR } // onChange event
                        />
                    </fieldset>
                    <fieldset>
                        <legend className="blocks-base-control__label">
                            { __( 'Text color', 'block-development-example' ) }
                        </legend>
                        <ColorPalette
                            onChange={ onChangeTextColor } // onChange event
                        />
                    </fieldset>
                </div>
            </InspectorControls>
            <TextControl
                value={ attributes.message }
                onChange={ ( val ) => setAttributes( { message: val } ) }
                style={ {
                    backgroundColor: attributes.bg_color,
                    color: attributes.text_color,
                } }
            />
        </div>
    );
}
```

See the [full block example](#) of the code above.

Block controls rendered in both the toolbar and sidebar will also be used when multiple blocks of the same type are selected.

For common customization settings including color, border, spacing customization and more, you can rely on [block supports](#) to provide the same functionality in a more efficient way.

Additional resources

- [Storybook for WordPress components](#)
- [@wordpress/block-editor](#)
- [@wordpress/components](#)
- [Inspector Controls](#)
- [BlockControls](#)

First published

December 18, 2023

Last updated

January 25, 2024

Edit article

[Improve it on GitHub: The block in the Editor”](#)

[Previous](#) [The block wrapper](#) [Previous: The block wrapper](#)

[Next](#) [Markup representation of a block](#) [Next: Markup representation of a block](#)

Markup representation of a block

[↑ Back to top](#)

When stored in the database or in templates as HTML files, blocks are represented using a [specific HTML grammar](#), which is technically valid HTML based on HTML comments that act as explicit block delimiters

These are some of the rules for the markup used to represent a block:

- All core block comments start with a prefix and the block name: `wp:blockname`
- For custom blocks, `blockname` is `namespace/blockname`
- The comment can be a single line, self-closing, or wrapper for HTML content.
- Custom block settings and attributes are stored as a JSON object inside the block comment.

Example: Markup representation of an image core block

```
<!-- wp:image -->
<figure class="wp-block-image"></figure>
<!-- /wp:image -->
```

The [markup representation of a block is parsed for the Block Editor](#) and the block's output for the front end:

- In the editor, WordPress parses this block markup, captures its data and loads its edit version
- In the front end, WordPress parses this block markup, captures its data and generates its final HTML markup

Whenever a block is saved, the `save` function, defined when the [block is registered in the client](#), is called to return the markup that will be saved into the database within the block delimiter's comment. If `save` is `null` (common case for blocks with dynamic rendering), only a single line block delimiter's comment is stored, along with any attributes

The Post Editor checks that the markup created by the `save` function is identical to the block's markup saved to the database:

- If there are any differences, the Post Editor triggers a [block validation error](#).
- Block validation errors usually happen when a block's `save` function is updated to change the markup produced by the block.
- A block developer can mitigate these issues by adding a [block deprecation](#) to register the change in the block.

The markup of a **block with dynamic rendering** is expected to change so the markup of these blocks is not saved to the database. What is saved in the database as representation of the block, for blocks with dynamic rendering, is a single line of HTML consisting on just the block delimiter's comment (including block attributes values). That HTML is not subject to the Post Editor's validation.

Example: Markup representation of a block with dynamic rendering (`save = null`) and attributes

```
<!-- wp:latest-posts {"postsToShow":4,"displayPostDate":true} /-->
```

Additional Resources

- [Data Flow and Data Format](#)
- [Static vs. dynamic blocks: What's the difference?](#)
- [Block deprecation – a tutorial](#)
- [Introduction to Templates > Block markup](#) | Theme Handbook

First published

December 19, 2023

Last updated

January 25, 2024

Edit article

[Improve it on GitHub: Markup representation of a block”](#)

[Previous](#) [The block in the Editor](#) [Previous: The block in the Editor](#)

[Next](#) [Static or Dynamic rendering of a block](#) [Next: Static or Dynamic rendering of a block](#)

Static or Dynamic rendering of a block

In this article

Table of Contents

- [Static rendering](#)
 - [How to define static rendering for a block](#)
- [Dynamic rendering](#)
 - [How to define dynamic rendering for a block](#)
 - [HTML representation of dynamic blocks in the database \(save\)](#)
- [Additional Resources](#)

[↑ Back to top](#)

The block's markup returned on the front end can be dynamically generated on the server when the block is requested from the client (dynamic blocks) or statically generated when the block is saved in the Block Editor (static blocks).

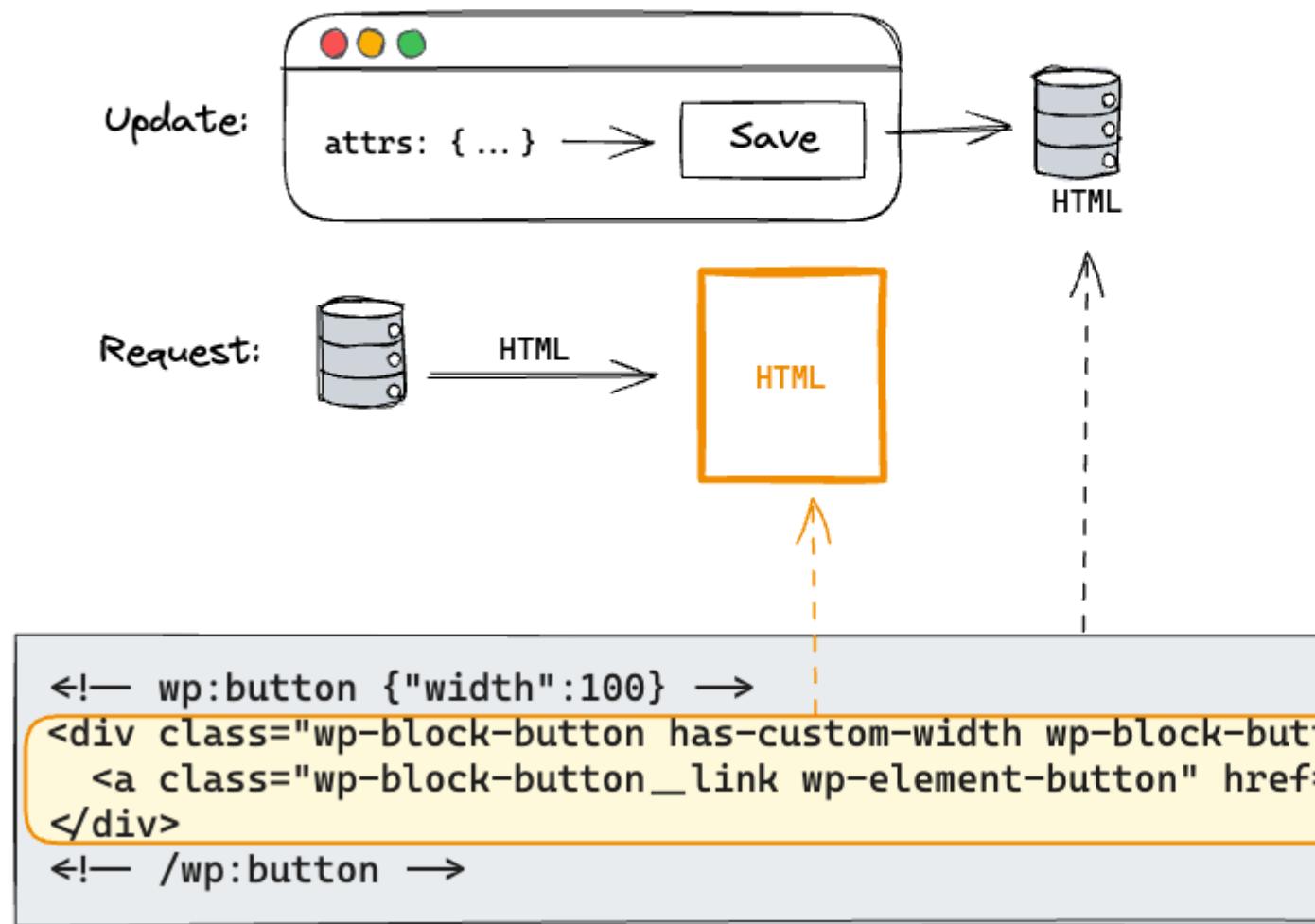
The post [Static vs. dynamic blocks: What's the difference?](#) provides a great introduction to static and dynamic blocks.

Static rendering

Static rendering

Generated at update-time

HTML generated with entity data



Blocks are considered “static” when they have “static rendering”, this is when their output for the front end is statically generated when saved to the database, as returned by their `save` functions.

Blocks have static rendering **when no dynamic rendering method has been defined (or is available) for the block**. In this case, the output for the front end will be taken from the [markup representation of the block in the database](#) that is returned by its `save` function when the block is saved in the Block Editor. This type of block is often called a “static block”.

How to define static rendering for a block

The `save` function, which can be defined when [registering a block on the client](#), determines the markup of the block that will be stored in the database when the content is saved and eventually returned to the front end when there’s a request. This markup is stored wrapped up in [unique](#)

[block delimiters](#) but only the markup inside these block indicators is returned as the markup to be rendered for the block on the front end.

To define static rendering for a block we define a `save` function for the block without any dynamic rendering method.

Example of static rendering of the preformatted core block

For example, the following [save function](#) of the preformatted core block...

```
import { RichText, useBlockProps } from '@wordpress/block-editor';

export default function save( { attributes } ) {
    const { content } = attributes;

    return (
        <pre { ...useBlockProps.save() }>
            <RichText.Content value={ content } />
        </pre>
    );
}
```

...generates the following markup representation of the block when `attributes.content` has the value "This is some preformatted text"...

```
<!-- wp:preformatted -->
<pre class="wp-block-preformatted">This is some preformatted text</pre>
<!-- /wp:preformatted -->
```

...and it will return the following markup for the block to the front end when there's a request.

```
<pre class="wp-block-preformatted">This is some preformatted text</pre>
```

Blocks with dynamic rendering can also define a markup representation of the block (via the `save` function) which can be processed in the server before returning the markup to the front end. If no dynamic rendering method is found, any markup representation of the block in the database will be returned to the front end.

The markup stored for a block can be modified before it gets rendered on the front end via hooks such as [render_block](#) or via `$render_callback`.

Some examples of core blocks with static rendering are:

- [separator](#) (see its `save` function)
- [spacer](#) (see its `save` function).
- [button](#) (see its `save` function).

[**Dynamic rendering**](#)

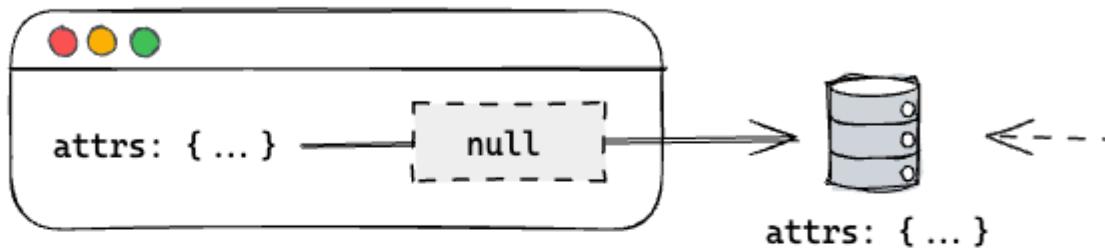
Blocks with dynamic rendering are blocks that **build their structure and content on the fly when the block is requested from the client**. This type of block is often called a “dynamic block”.

Dynamic rendering

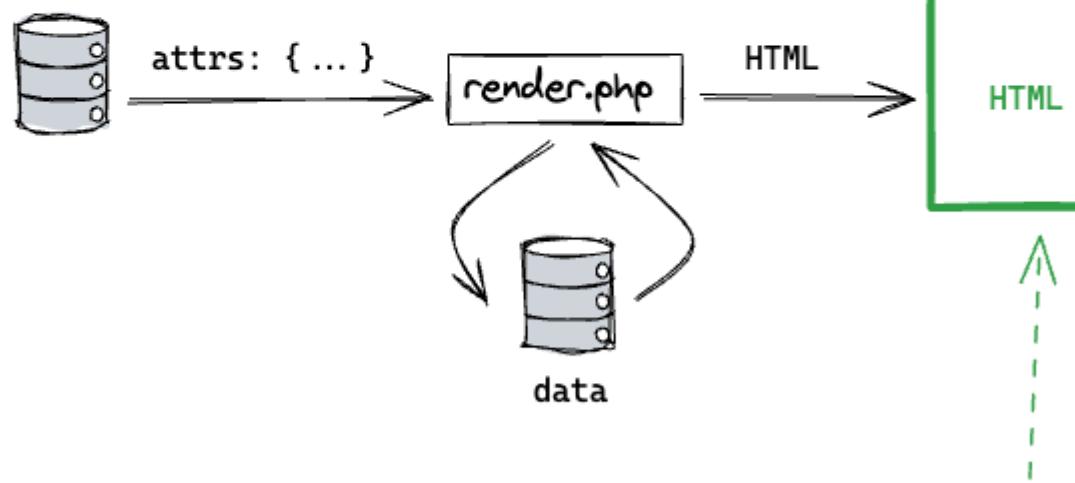
Generated at request-time

HTML generated
with other data

Update:



Request:



```
<h1 class="wp-block-site-title">
  <a href="https://mysite.com" target="_blank" rel="ho
</h1>
```

There are some common use cases for dynamic blocks:

1. **Blocks where content should change even if a post has not been updated.** An example is the [latest-posts core block](#), which will update its content on request time, everywhere it is used after a new post is published.
2. **Blocks where updates to the markup should be immediately shown on the front end of the website.** For example, if you update the structure of a block by adding a new class, adding an HTML element, or changing the layout in any other way, using a dynamic block ensures those changes are applied immediately on all occurrences of that block across the site. If a dynamic block is not used then when block code is updated, Gutenberg's [validation process](#) generally applies, causing users to see the validation message: “This block appears to have been modified externally”.

How to define dynamic rendering for a block

A block can define dynamic rendering in two main ways:

1. Via the `render_callback` argument that can be passed to the [register_block_type\(\) function](#).
2. Via a separate PHP file (usually named `render.php`) which path can be defined at the [render property of the block.json](#).

Both of these ways to define the block's dynamic rendering receive the following data:

- `$attributes` – The array of attributes for this block.
- `$content` – Rendered block output (markup of the block as stored in the database).
- `$block` – The instance of the [WP_Block](#) class that represents the block being rendered ([metadata of the block](#)).

Example of dynamic rendering of the site-title core block

For example, the [site-title](#) core block with the following function registered as [render_callback](#)...

```
function render_block_core_site_title( $attributes ) {  
    $site_title = get_bloginfo( 'name' );  
    if ( ! $site_title ) {  
        return;  
    }  
  
    $tag_name = 'h1';  
    $classes = empty( $attributes['textAlign'] ) ? '' : "has-text-align-{$attributes['textAlign']}";  
    if ( isset( $attributes['style']['elements']['link']['color']['text'] ) )  
        $classes .= ' has-link-color';  
  
    if ( isset( $attributes['level'] ) ) {  
        $tag_name = 0 === $attributes['level'] ? 'p' : 'h' . (int) $attributes['level'];  
    }  
  
    if ( $attributes['isLink'] ) {  
        $aria_current = is_home() || ( is_front_page() && 'page' === get_option( 'page_on_front' ) );  
        $link_target = ! empty( $attributes['linkTarget'] ) ? $attributes['linkTarget'] : $site_title;  
  
        $site_title = sprintf(  
            '<a href="%1$s" target="%2$s" rel="home">%3$s>%4$s</a>',  
            esc_url( home_url() ),  
            esc_attr( $link_target ),  
            $aria_current,  
            esc_html( $site_title )  
        );  
    }  
    $wrapper_attributes = get_block_wrapper_attributes( array( 'class' =>  
  
        return sprintf(  
            '<%1$s %2$s>%3$s</%1$s>',  
            $tag_name,
```

```
$wrapper_attributes,  
    // already pre-escaped if it is a link.  
    $attributes['isLink'] ? $site_title : esc_html( $site_title )  
);  
}
```

... generates the following markup representation of the block in the database (as [there's no save function defined for this block](#))...

```
<!-- wp:site-title -->
```

...and it could generate the following markup for the block to the front end when there's a request (depending on the specific values on the server at request time).

```
<h1 class="wp-block-site-title"><a href="https://www.wp.org" target="_self"
```

[**HTML representation of dynamic blocks in the database \(save\)**](#)

For dynamic blocks, the `save` callback function can return just `null`, which tells the editor to save only the block delimiter comment (along with any existing [block attributes](#)) to the database. These attributes are then passed into the server-side rendering callback, which will determine how to display the block on the front end of your site. **When `save` is `null`, the Block Editor will skip the [block markup validation process](#)**, avoiding issues with frequently changing markup.

Blocks with dynamic rendering can also save an HTML representation of the block as a backup. If you provide a server-side rendering callback, the HTML representing the block in the database will be replaced with the output of your callback, but will be rendered if your block is deactivated (the plugin that registers the block is uninstalled) or your render callback is removed.

In some cases, the block saves an HTML representation of the block and uses a dynamic rendering to fine-tune this markup if some conditions are met. Some examples of core blocks using this approach are:

- The [cover](#) block saves a [full HTML representation of the block in the database](#). This markup is processed via a [render callback](#) when requested to do some PHP magic that dynamically [injects the featured image if the “use featured image” setting is enabled](#).
- The [image](#) block also saves [its HTML representation in the database](#) and processes it via a [render callback](#) when requested to [add some attributes to the markup](#) if some conditions are met.

If you are using [InnerBlocks](#) in a dynamic block, you will need to save the `InnerBlocks` in the `save` callback function using `<InnerBlocks.Content/>`.

[**Additional Resources**](#)

- [Static vs. dynamic blocks: What’s the difference?](#)
- [Block deprecation – a tutorial](#)

First published

January 8, 2024

Last updated

January 25, 2024

[Edit article](#)

[Improve it on GitHub: Static or Dynamic rendering of a block”](#)

[Previous](#) [Markup representation of a block](#) [Previous: Markup representation of a block](#)
[Next](#) [Working with Javascript for the Block Editor](#) [Next: Working with Javascript for the Block Editor](#)

Working with Javascript for the Block Editor

In this article

[Table of Contents](#)

- [JavaScript build process](#)
- [Javascript without a build process](#)
- [Additional resources](#)

[↑ Back to top](#)

A JavaScript Build Process is recommended for most cases when working with Javascript for the Block Editor. With a build process, you'll be able to work with ESNext and JSX (among others) syntaxes and features in your code while producing code ready for the majority of the browsers.

[JavaScript build process](#)

“[ESNext](#)” is a dynamic name that refers to Javascript’s latest syntax and features. “[JSX](#)” is a custom syntax extension to JavaScript, created by React project, that allows you to write JavaScript using a familiar HTML tag-like syntax.

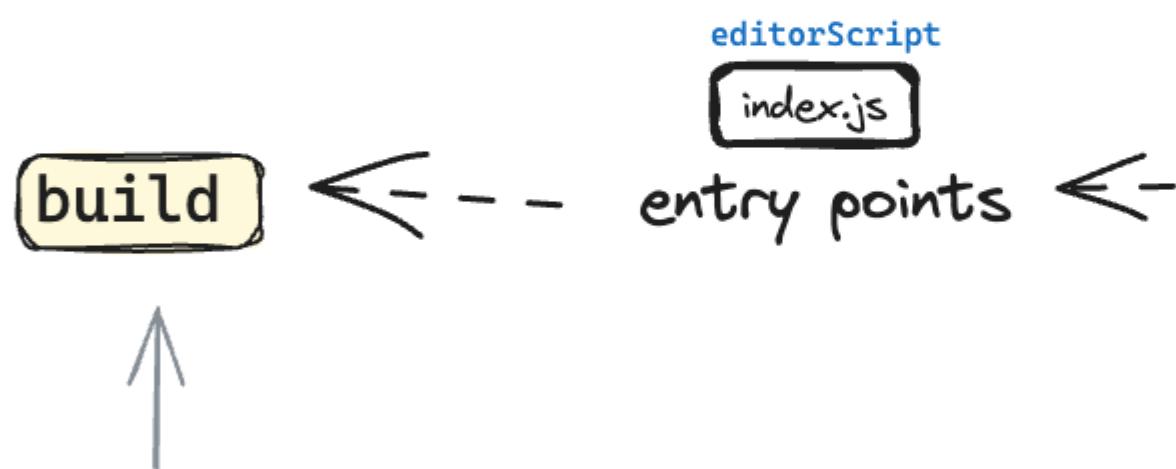
Browsers cannot interpret or run ESNext and JSX syntaxes, so a transformation step is needed to convert these syntaxes to code that browsers can understand.

“[webpack](#)” is a pluggable tool that processes JavaScript and creates a compiled bundle that runs in a browser. “[babel](#)” transforms JavaScript from one format to another. Babel is a webpack plugin to transform ESNext and JSX to production-ready JavaScript.

[@wordpress/scripts](#) package abstracts these libraries away to standardize and simplify development, so you won’t need to handle the details for configuring webpack or babel. Check the [Get started with wp-scripts](#) intro guide.

Among other things, with `wp-scripts` package you can use Javascript modules to distribute your code among different files and get a few bundled files at the end of the build process (see [example](#)).

The source directory can be customised us



The output directory can be customised us

With the [proper package.json scripts](#) you can launch the build process with wp-scripts in production and development mode:

- `npm run build` for “production” mode build – This process [minifies the code](#) so it downloads faster in the browser.
- `npm run start` for “development” mode build – This process does not minify the code of the bundled files, provides [source maps files](#) for them, and additionally continues a running process to watch the source file for more changes and rebuilds as you develop.

You can [provide your own custom webpack.config.js](#) to wp-scripts to customize the build process to suit your needs

Javascript without a build process

Using Javascript without a build process may be another good option for code developments with few requirements (especially those not requiring JSX).

Without a build process, you access the methods directly from the `wp` global object and must enqueue the script manually. [WordPress Javascript packages](#) can be accessed through the [wp global variable](#) but every script that wants to use them through this `wp` object is responsible for adding [the handle of that package](#) to the dependency array when registered.

So, for example if a script wants to register a block variation using the `registerBlockVariation` method out of the [“blocks” package](#), the `wp-blocks` handle would need to get added to the dependency array to ensure that `wp.blocks.registerBlockVariation` is defined when the script tries to access it (see [example](#)).

Try running `wp.data.select('core/editor').getBlocks()` in your browser’s dev tools while editing a post or a site. The entire editor is available from the console.

Use [enqueue_block_editor_assets](#) hook coupled with the standard [wp_enqueue_script](#) (and [wp_register_script](#)) to enqueue javascript assets for the Editor with access to these packages via `wp` (see [example](#)). Refer to [Enqueueing assets in the Editor](#) for more info.

Additional resources

- [Package Reference](#)
- [Get started with wp-scripts](#)
- [Enqueueing assets in the Editor](#)
- [WordPress Packages handles](#)
- [Javascript Reference](#) | MDN Web Docs
- [block-development-examples](#) | GitHub repository
- [block-theme-examples](#) | GitHub repository
- [How webpack and WordPress packages interact](#) | Developer Blog
- [Build Process Diagram](#)

First published

November 28, 2023

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Working with Javascript for the Block Editor”](#)

[Previous Static or Dynamic rendering of a block](#) [Previous: Static or Dynamic rendering of a block](#)

[Next Glossary](#) [Next: Glossary](#)

Glossary

In this article

Table of Contents

- [Attribute sources](#)
- [Attributes](#)
- [Block](#)
- [Block Styles](#)
- [Block Supports](#)
- [Block Theme](#)
- [Block categories](#)
- [Block ~Inserter~ Library](#)
- [Block name](#)
- [Block Templates](#)
- [Block Template Parts](#)
- [Block type](#)
- [Classic block](#)
- [Dynamic block](#)
- [Full Site Editing](#)
- [Global Styles](#)
- [Inspector](#)
- [Local Styles](#)
- [Navigation Block](#)
- [Patterns](#)
- [Post settings](#)
- [Query Block](#)
- [Reusable block](#)
- [RichText](#)
- [Serialization](#)
- [Settings Sidebar](#)
- [Site Editor](#)
- [Static block](#)
- [Template Editing Mode](#)
- [Theme Blocks](#)
- [TinyMCE](#)
- [Toolbar](#)

[↑ Back to top](#)

Attribute sources

An object describing the attributes shape of a block. The keys can be named as most appropriate to describe the state of a block type. The value for each key is a function which describes the strategy by which the attribute value should be extracted from the content of a saved post's content. When processed, a new object is created, taking the form of the keys defined in the attribute sources, where each value is the result of the attribute source function.

Attributes

The object representation of the current state of a block in post content. When loading a saved post, this is determined by the attribute sources for the block type. These values can change over time during an editing session when the user modifies a block, and are used when determining how to serialize the block.

Block

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.

Block Styles

The CSS styles that are part of the block, either via its stylesheet or via the block markup itself. For example, a class attached to the block markup is considered block styles.

Compare to [Global Styles](#). In contraposition to Global Styles, block styles are sometimes referred to as [Local Styles](#).

Learn more about [Block Styles](#).

Block Supports

An API for blocks to declare what features they support. By declaring support for a feature, the API would add additional [attributes](#) to the block and matching UI controls for most of the existing block supports.

See [Block Supports reference documentation](#) for a deep dive into the API.

Block Theme

A theme built in block forward way that allows Full Site Editing to work. The core of a block theme are its block templates and block template parts. To date, block theme templates have been HTML files of block markup that map to templates from the standard WordPress template hierarchy.

Block categories

These are not a WordPress taxonomy, but instead used internally to sort blocks in the Block Library.

Block ~Inserter~ Library

Primary interface for selecting from the available blocks, triggered by plus icon buttons on Blocks or in the top-left of the editor interface.

Block name

A unique identifier for a block type, consisting of a plugin-specific namespace and a short label describing the block's intent. e.g. `core/image`

Block Templates

A template is a pre-defined arrangement of blocks, possibly with predefined attributes or placeholder content. You can provide a template for a post type, to give users a starting point when creating a new piece of content, or inside a custom block with the `InnerBlocks` component. At their core, templates are simply HTML files of block markup that map to templates from the standard WordPress template hierarchy, for example index, single or archive. This helps control the front-end defaults of a site that are not edited via the Page Editor or the Post Editor. See the [templates documentation](#) for more information.

Block Template Parts

Building on Block Templates, these parts help set structure for reusable items like a Footer or Header that one typically sees in a WordPress site. They are primarily site structure and are never to be mixed with the post content editor. With Full Site Editing and block based themes, users can create their own arbitrary Template Parts, save those in the database for their site, and re-use them throughout their site. Template parts are equivalent – in blocks – of theme template parts. They are generally defined by a theme first, carry some semantic meaning (could be swapped between themes such as a header), and can only be inserted in the site editor context (within “templates”).

Block type

In contrast with the blocks composing a particular post, a block type describes the blueprint by which any block of that type should behave. So while there may be many images within a post, each behaves consistent with a unified image block type definition.

Classic block

A block which embeds the TinyMCE editor as a block, TinyMCE was the base of the previous core editor. Older content created prior to the block editor will be loaded in to a single Classic block.

Dynamic block

A type of block where the content of which may change and cannot be determined at the time of saving a post, instead calculated any time the post is shown on the front of a site. These blocks may save fallback content or no content at all in their JavaScript implementation, instead deferring to a PHP block implementation for runtime rendering.

Full Site Editing

This refers to a collection of features that ultimately allows users to edit their entire website using blocks as the starting point. This feature set includes everything from block patterns to global styles to templates to design tools for blocks (and more). First released in WordPress 5.9.

Global Styles

The CSS styles generated by WordPress and enqueued as an embedded stylesheet in the front end of the site. The stylesheet ID is `global-styles-inline-css`. The contents of this stylesheet come from the default `theme.json` of WordPress, the theme's `theme.json`, and the styles provided by the user via the global styles sidebar in the site editor.

See [theme.json reference docs](#), the [how to guide](#), and an introduction to [styles in the block editor](#).

Compare to [block styles](#).

Inspector

Deprecated term. See [Settings Sidebar](#).

Local Styles

See [Block Styles](#).

Navigation Block

A block that allows you to edit a site's navigation menu, both in terms of structure and design.

Patterns

Patterns are predefined layouts of blocks that can be inserted as starter content that are meant to be changed by the user every time. Once inserted, they exist as a local save and are not global.

Post settings

A sidebar region containing metadata fields for the post, including scheduling, visibility, terms, and featured image.

Query Block

A block that replicates the classic [WP_Query](#) and allows for further customization with additional functionality.

Reusable block

A block that is saved and then can be shared as a reusable, repeatable piece of content.

RichText

A common component enabling rich content editing including bold, italics, hyperlinks, etc.

Serialization

The process of converting a block's attributes object into HTML markup, which occurs each time a block is edited.

Settings Sidebar

The panel on the right that contains the document and block settings. The sidebar is toggled using the Settings gear icon. Block settings are shown when a block is selected, otherwise document settings are shown.

Site Editor

The cohesive experience that allows you to directly edit and navigate between various templates, template parts, styling options, and more.

Static block

A type of block where the content of which is known at the time of saving a post. A static block will be saved with HTML markup directly in post content.

Template Editing Mode

A scaled down direct editing experience allowing you to edit/change/create the template a post/page uses.

Theme Blocks

Blocks that accomplish everything possible in traditional templates using template tags (ex: Post Author Block). A full list can be found [here](#).

TinyMCE

[TinyMCE](#) is a web-based JavaScript WYSIWYG (What You See Is What You Get) editor.

Toolbar

A set of button controls. In the context of a block, usually referring to the toolbar of block controls shown above the selected block.

First published

March 10, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Glossary”](#)

[Previous](#) [Working with Javascript for the Block Editor](#) [Previous: Working with Javascript for the Block Editor](#)

[Next Frequently Asked Questions](#) [Next: Frequently Asked Questions](#)

Frequently Asked Questions

In this article

[Table of Contents](#)

- [The Gutenberg Project](#)
 - [What is Gutenberg?](#)
 - [What’s on the roadmap long term?](#)
 - [When was Gutenberg started?](#)
 - [When was Gutenberg merged into WordPress?](#)
 - [WordPress is already the world’s most popular publishing platform. Why change the editor at all?](#)
- [The Editing Experience](#)
 - [What are “blocks” and why are we using them?](#)
 - [What is the writing experience like?](#)
 - [Is Gutenberg built on top of TinyMCE?](#)
 - [Are there Keyboard Shortcuts for Gutenberg?](#)
 - [Does Gutenberg support columns?](#)
 - [Does Gutenberg support nested blocks?](#)
 - [Does drag and drop work for rearranging blocks?](#)
- [The Development Experience](#)
 - [How do I make my own block?](#)
 - [Does Gutenberg involve editing posts/pages in the front end?](#)
 - [Given Gutenberg is built in JavaScript, how do old meta boxes \(PHP\) work?](#)
 - [How can plugins extend the Gutenberg UI?](#)
 - [Are Custom Post Types still supported?](#)
- [Styles](#)
 - [Can themes style blocks?](#)
 - [How do block styles work in both the front-end and back-end?](#)
 - [What are block variations? Are they the same as block styles?](#)

- [How do editor styles work?](#)
- [Compatibility](#)
 - [What browsers does Gutenberg support?](#)
 - [Should I be concerned that Gutenberg will make my plugin obsolete?](#)
 - [Is it possible to opt out of Gutenberg for my site?](#)
 - [How do custom TinyMCE buttons work in Gutenberg?](#)
 - [How do shortcodes work in Gutenberg?](#)
 - [Should I move shortcodes to content blocks?](#)
- [Miscellaneous](#)
 - [Is Gutenberg made to be properly accessible?](#)
 - [How is data stored? I've seen HTML comments. What is their purpose?](#)
 - [How can I parse the post content back out into blocks in PHP or JS?](#)

[↑ Back to top](#)

What follows is a set of questions that have come up from the last few years of Gutenberg development. If you have any questions you'd like to have answered and included here, [just open up a GitHub issue](#) with your question. We'd love the chance to answer and provide clarity to questions we might not have thought to answer. For a look back historically, please see Matt's November 2018 post [WordPress 5.0: A Gutenberg FAQ](#).

[The Gutenberg Project](#)

[What is Gutenberg?](#)

“Gutenberg” is the name of the project to create a new editor experience for WordPress — contributors have been working on it since January 2017 and it’s one of the most significant changes to WordPress in years. It’s built on the idea of using “blocks” to write and design posts and pages. This will serve as the foundation for future improvements to WordPress, including blocks as a way not just to design posts and pages, but also entire sites. The overall goal is to simplify the first-time user experience of WordPress — for those who are writing, editing, publishing, and designing web pages. The editing experience is intended to give users a better visual representation of what their post or page will look like when they hit publish. Originally, this was the kickoff goal:

The editor will endeavour to create a new page and post building experience that makes writing rich posts effortless, and has “blocks” to make it easy what today might take shortcodes, custom HTML, or “mystery meat” embed discovery.

Key takeaways include the following points:

- Authoring richly laid-out posts is a key strength of WordPress.
- By embracing blocks as an interaction paradigm, we can unify multiple different interfaces into one. Instead of learning how to write shortcodes and custom HTML, or pasting URLs to embed media, there’s a common, reliable flow for inserting any kind of content.
- “Mystery meat” refers to hidden features in software, features that you have to discover. WordPress already supports a large number of blocks and 30+ embeds, so let’s surface them.

Gutenberg is developed on [GitHub](#) under the WordPress organization. The block editor has been available in core WordPress since 5.0. If you want to test upcoming features from Gutenberg project, it is [available in the plugin repository](#).

What's on the roadmap long term?

There are four phases of Gutenberg which you can see on the [official WordPress roadmap](#). As of writing this, we're currently in phase 2:

1. Easier Editing — Already available in WordPress since 5.0, with ongoing improvements.
2. Customization — Full Site editing, Block Patterns, Block Directory, Block based themes.
3. Collaboration — A more intuitive way to co-author content
4. Multi-lingual — Core implementation for Multi-lingual sites

When was Gutenberg started?

The editor focus started in early 2017 with the first three months spent designing, planning, prototyping, and testing prototypes, to help us inform how to approach this project. The first plugin was launched during WordCamp Europe in June 2017.

When was Gutenberg merged into WordPress?

Gutenberg was first merged into [WordPress 5.0](#) in December 2018. See [the versions in WordPress page](#) for a complete list of Gutenberg plugin versions merged into WordPress core releases.

WordPress is already the world's most popular publishing platform. Why change the editor at all?

The Editor is where most of the action happens in WordPress's daily use, and it was a place where we could polish and perfect the block experience in a contained environment. Further, as an open-source project, we believe that it is critical for WordPress to continue to innovate and keep working to make the core experience intuitive and enjoyable for all users. As a community project, Gutenberg has the potential to do just that, and we're excited to pursue this goal together. If you'd like to test, contribute, or offer feedback, we welcome you to [share what you find on GitHub](#).

The Editing Experience

What are “blocks” and why are we using them?

The classic WordPress editor is an open text window—it's always been a wonderful blank canvas for writing, but when it comes to building posts and pages with images, multimedia, embedded content from social media, polls, and other elements, it required a mix of different approaches that were not always intuitive:

- Media library/HTML for images, multimedia and approved files.
- Pasted links for embeds.
- Shortcodes for specialized assets from plugins.
- Featured images for the image at the top of a post or page.
- Excerpts for subheadings.
- Widgets for content on the side of a page.

As we thought about these uses and how to make them obvious and consistent, we began to embrace the concept of “blocks.” All of the above items could be blocks: easy to search and understand, and easy to dynamically shift around the page. The block concept is very powerful, and when designed thoughtfully, can offer an outstanding editing and publishing experience.

Ultimately, the idea with blocks is to create a new common language across WordPress, a new way to connect users to plugins, and replace a number of older content types — things like shortcodes and widgets — that one usually has to be well-versed in the idiosyncrasies of WordPress to understand.

What is the writing experience like?

Our goal with Gutenberg is not just to create a seamless post- and page-building experience. We also want to ensure that it provides a seamless writing experience. To test this out yourself, [head to this demo and give it a try!](#)

Is Gutenberg built on top of TinyMCE?

No. [TinyMCE](#) is only used for the “Classic” block.

Are there Keyboard Shortcuts for Gutenberg?

Yes. There are a lot! There is a help modal showing all available keyboard shortcuts.

You can see the whole list going to the top right corner menu of the new editor and clicking on “Keyboard Shortcuts” (or by using the keyboard shortcut Shift+Alt+H on Linux/Windows and ⌘H on macOS).

This is the canonical list of keyboard shortcuts:

Editor shortcuts

Shortcut description	Linux/Windows shortcut	macOS shortcut
Display keyboard shortcuts.	Shift+Alt+H	⌘H
Save your changes.	Ctrl+S	⌘S
Undo your last changes.	Ctrl+Z	⌘Z
Redo your last undo.	Ctrl+Shift+Z	⇧ ⌘Z
Show or hide the Settings sidebar.	Ctrl+Shift+,	⇧ ⌘,
Open the list view menu.	Shift+Alt+0	⌘0
Navigate to the next part of the editor.	Ctrl+`	^`
Navigate to the previous part of the editor.	Ctrl+Shift+`	^↑ `
Navigate to the next part of the editor (alternative).	Ctrl+Alt+N	⌘N
Navigate to the previous part of the editor (alternative).	Ctrl+Alt+P	⌘P
Navigate to the nearest toolbar.	Alt+F10	F10
Switch between visual editor and code editor.	Ctrl+Shift+Alt+M	⇧ ⌘M
Toggle fullscreen mode.	Ctrl+Alt+Shift+F	⇧ ⌘F

Selection shortcuts

Shortcut description	Linux/Windows shortcut	macOS shortcut
Select all text when typing. Press again to select all blocks.	Ctrl+A	⌘A
Clear selection.	Esc	Esc
Select text across multiple blocks.	Shift+Arrow (←, ↑, →, ↓)	Shift+Arrow (←, ↑, →, ↓)

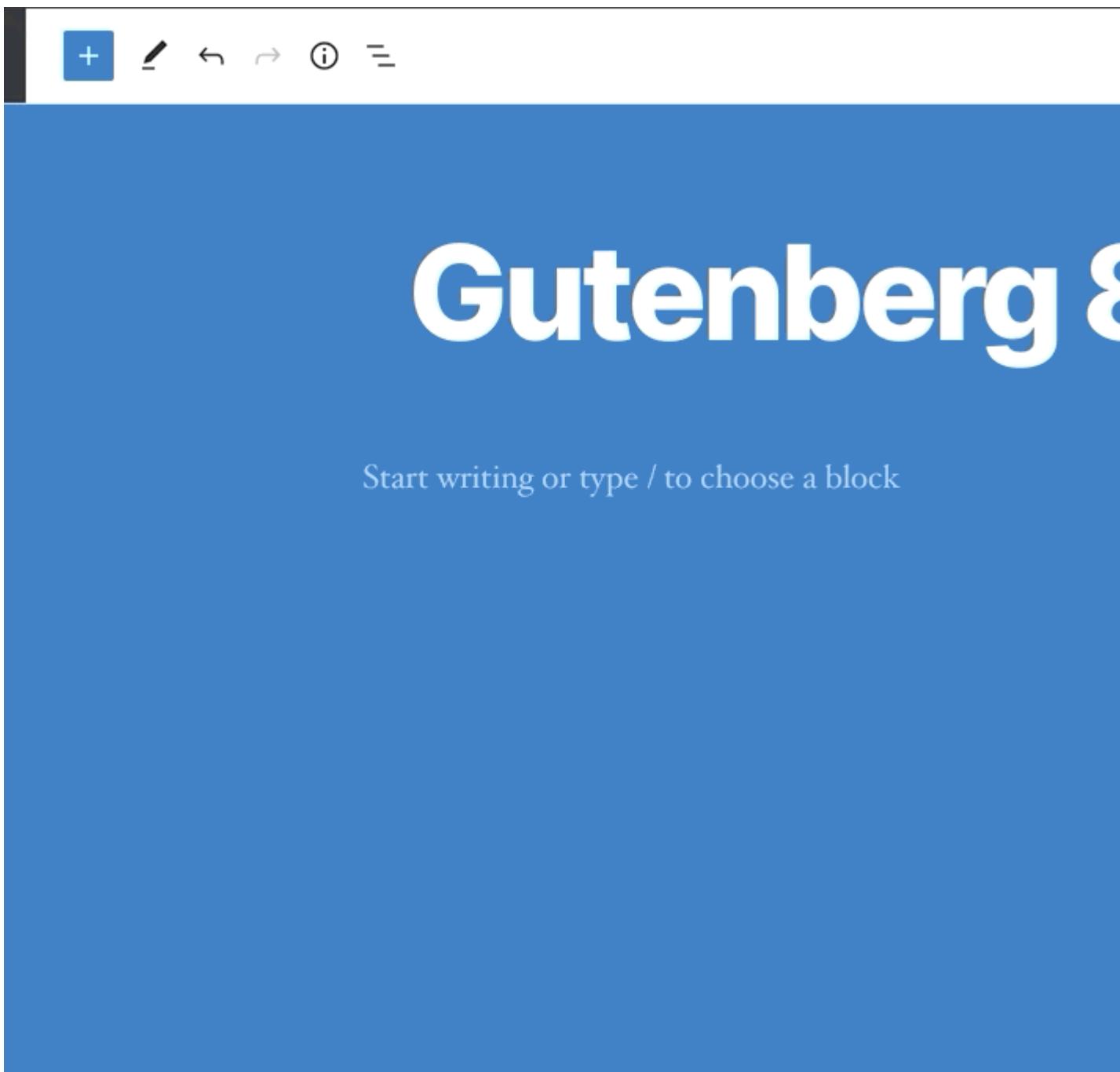
Block shortcuts

Shortcut description	Linux/Windows shortcut	macOS shortcut
Duplicate the selected block(s).	Ctrl+Shift+D	↑ ⌘D
Remove the selected block(s).	Shift+Alt+Z	^⌫Z
Insert a new block before the selected block(s).	Ctrl+Alt+T	⌘T
Insert a new block after the selected block(s).	Ctrl+Alt+Y	⌘Y
Move the selected block(s) up.	Ctrl+Alt+Shift+T	⌘ ↑ T
Move the selected block(s) down.	Ctrl+Alt+Shift+Y	⌘ ↑ Y
Change the block type after adding a new paragraph.	/	/
Remove multiple selected blocks.	delbackspace	delbackspace

Text formatting

Shortcut description	Linux/Windows shortcut	macOS shortcut
Make the selected text bold.	Ctrl+B	⌘B
Make the selected text italic.	Ctrl+I	⌘I
Underline the selected text.	Ctrl+U	⌘U
Convert the selected text into a link.	Ctrl+K	⌘K
Remove a link.	Ctrl+Shift+K	↑ ⌘K
Add a strikethrough to the selected text.	Shift+Alt+D	^⌫D
Display the selected text in a monospaced font.	Shift+Alt+X	^⌫X

Here is a brief animation illustrating how to find and use the keyboard shortcuts:



Does Gutenberg support columns?

Yes, a columns block is available in Gutenberg.

Does Gutenberg support nested blocks?

Yes, it is supported. You can have multiple levels of nesting – blocks within blocks within blocks. See the [Nested Block Tutorial](#) for more information.

Does drag and drop work for rearranging blocks?

Yes, you can drag and drop blocks to rearrange their order.

The Development Experience

How do I make my own block?

The best place to start is the [Create a Block Tutorial](#).

Does Gutenberg involve editing posts/pages in the front end?

No, we are designing Gutenberg primarily as a replacement for the post and page editing screens. That said, front-end editing is often confused with an editor that looks exactly like the front end. And that is something that Gutenberg will allow as themes customize individual blocks and provide those styles to the editor. Since content is designed to be distributed across so many different experiences—from desktop and mobile to full-text feeds and syndicated article platforms—we believe it's not ideal to create or design posts from just one front-end experience.

Given Gutenberg is built in JavaScript, how do old meta boxes (PHP) work?

See the [Meta Box Tutorial](#) for more information on using Meta boxes with the new block editor.

How can plugins extend the Gutenberg UI?

The main extension point we want to emphasize is creating new blocks. Blocks are added to the block editor using plugins, see the [Create a Block Tutorial](#) to get started.

Are Custom Post Types still supported?

Indeed. There are multiple ways in which custom post types can leverage Gutenberg. The plan is to allow them to specify the blocks they support, as well as defining a default block for the post type. It's not currently the case, but if a post type disables the content field, the “advanced” section at the bottom would fill the page.

Styles

Can themes style blocks?

Yes. Blocks can provide their own styles, which themes can add to or override, or they can provide no styles at all and rely fully on what the theme provides.

How do block styles work in both the front-end and back-end?

Blocks are able to provide base structural CSS styles, and themes can add styles on top of this. Some blocks, like a Separator (`<hr />`), likely don't need any front-end styles, while others, like a Gallery, need a few.

Other features, like the new *wide* and *full-wide* alignment options, are simply CSS classes applied to blocks that offer this alignment. We are looking at how a theme can opt into this feature, for example using `add_theme_support`.

This is currently a work in progress and we recommend reviewing the [block based theme documentation](#) to learn more.

What are block variations? Are they the same as block styles?

No, [block variations](#) are different versions of a single base block, sharing a similar functionality but with slight differences in their implementation or settings (attributes, InnerBlocks, etc.). Block variations are transparent for users, and once there is a registered block variation, it will appear as a new block. For example, the embed block registers different block variations to embed content from specific providers.

Meanwhile, [block styles](#) allow you to provide alternative styles to existing blocks, and they work by adding a `className` to the block's wrapper. Once a block has registered block styles, a block style selector will appear in its sidebar so that users can choose among the different registered styles.

How do editor styles work?

Regular editor styles are opt-in and work as is in most cases. Themes can also load extra stylesheets by using the following hook:

```
function gutenbergtheme_editor_styles() {  
    wp_enqueue_style( 'gutenbergtheme-blocks-style', get_template_directory() );  
}  
add_action( 'enqueue_block_editor_assets', 'gutenbergtheme_editor_styles'
```

See: [Editor Styles](#)

Compatibility

What browsers does Gutenberg support?

Gutenberg works in modern browsers.

The [list of supported browsers can be found in the Make WordPress handbook](#). The term “modern browsers” generally refers to the *current and previous two versions* of each major browser.

Since WordPress 5.8, Gutenberg no longer supports any version of Internet Explorer.

Should I be concerned that Gutenberg will make my plugin obsolete?

The goal of Gutenberg is not to put anyone out of business. It’s to evolve WordPress so there’s more business to be had in the future, for everyone.

Aside from enabling a rich post and page building experience, a meta goal is to *move WordPress forward* as a platform. Not only by modernizing the UI, but by modernizing the foundation.

We realize it’s a big change. We also think there will be many new opportunities for plugins. WordPress is likely to ship with a range of basic blocks, but there will be plenty of room for highly tailored premium plugins to augment existing blocks or add new blocks to the mix.

Is it possible to opt out of Gutenberg for my site?

There is a “Classic” block, which is virtually the same as the current editor, except in block form.

There is also the [Classic Editor plugin](#) which restores the previous editor, see the plugin for more information. The WordPress Core team has committed to supporting the Classic Editor plugin [until December 2021](#).

How do custom TinyMCE buttons work in Gutenberg?

Custom TinyMCE buttons still work in the “Classic” block, which is a block version of the classic editor you know today.

Gutenberg comes with a new universal inserter tool, which gives you access to every block available, searchable, sorted by recency and categories. This inserter tool levels the playing field for every plugin that adds content to the editor, and provides a single interface to learn how to use.

How do shortcodes work in Gutenberg?

Shortcodes continue to work as they do now.

However we see the block as an evolution of the [shortcode]. Instead of having to type out code, you can use the universal inserter tray to pick a block and get a richer interface for both configuring the block and previewing it. We would recommend people eventually upgrade their shortcodes to be blocks.

Should I move shortcodes to content blocks?

We think so for a variety of reasons including but not limited to:

- Blocks have visual editing built-in which creates a more rich, dynamic experience for building your site.
- Blocks are simply html and don't persist things the browser doesn't understand on the front-end. In comparison, if you disable a plugin that powers a shortcode, you end up with strange visuals on the front-end (often just showing the shortcode in plain text).
- Blocks will be discovered more readily with the launch of the block directory in a way shortcodes never could be allowing for more people to get more functionality.

Ultimately, Blocks are designed to be visually representative of the final look, and, with the launch of the Block Directory in 5.5, they will become the expected way in which users will discover and insert content in WordPress.

Miscellaneous

Is Gutenberg made to be properly accessible?

Accessibility is not an afterthought. Not every aspect of Gutenberg is accessible at the moment. You can check logged issues [here](#). We understand that WordPress is for everyone, and that accessibility is about inclusion. This is a key value for us.

If you would like to contribute to the accessibility of Gutenberg, we can always use more people to test and contribute.

[How is data stored? I've seen HTML comments. What is their purpose?](#)

Our approach—as outlined in [the technical overview introduction](#)—is to augment the existing data format in a way that doesn’t break the decade-and-a-half-fabric of content WordPress provides. In other terms, this optimizes for a format that prioritizes human readability (the HTML document of the web) and easy-to-render-anywhere over a machine convenient file (JSON in post-meta) that benefits the editing context primarily.

This also [gives us the flexibility](#) to store those blocks that are inherently separate from the content stream (reusable pieces like widgets or small post type elements) elsewhere, and just keep token references for their placement.

We suggest you look at the [Gutenberg key concepts](#) to learn more about how this aspect of the project works.

[How can I parse the post content back out into blocks in PHP or JS?](#)

In JS:

```
var blocks = wp.blocks.parse( postContent );
```

In PHP:

```
$blocks = parse_blocks( $post_content );
```

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Frequently Asked Questions”](#)

[Previous Glossary Previous: Glossary](#)
[Next How-to Guides Next: How-to Guides](#)

How-to Guides

In this article

Table of Contents

- [Creating blocks](#)
- [Extending blocks](#)
- [Extending the Editor UI](#)
- [Meta boxes](#)
- [Theme support](#)
- [Autocomplete](#)
- [Block parsing and serialization](#)

[↑ Back to top](#)

The new editor is highly flexible, like most of WordPress. You can build custom blocks, modify the editor's appearance, add special plugins, and much more.

Creating blocks

The editor is about blocks, and the main extensibility API is the Block API. It allows you to create your own static blocks, [Dynamic Blocks](#) (rendered on the server) and also blocks capable of saving data to Post Meta for more structured content.

If you want to learn more about block creation, see the [Create a Block tutorial](#) for the best place to start.

Extending blocks

It is also possible to modify the behavior of existing blocks or even remove them completely using filters.

Learn more in the [Block Filters](#) section.

Specifically for Query Loop block, besides the available filters, there are more ways to extend it and create bespoke versions of it. Learn more in the [Extending the Query Loop block](#) section.

Extending the Editor UI

Extending the editor UI can be accomplished with the `registerPlugin` API, allowing you to define all your plugin's UI elements in one place.

Refer to the [Plugins](#) and [Edit Post](#) section for more information.

You can also filter certain aspects of the editor; this is documented on the [Editor Filters](#) page.

Meta boxes

Porting PHP meta boxes to blocks or sidebar plugins is highly encouraged, learn how in the [meta box](#) and [sidebar plugin](#) guides.

Theme support

By default, blocks provide their styles to enable basic support for blocks in themes without any change. Themes can add/override these styles, or rely on defaults.

There are some advanced block features which require opt-in support in the theme. See [theme support](#) and [how to filter global styles](#).

Autocomplete

Autocompleters within blocks may be extended and overridden. Learn more about the [autocomplete](#) filters.

Block parsing and serialization

Posts in the editor move through a couple of different stages between being stored in `post_content` and appearing in the editor. Since the blocks themselves are data structures that live in memory it takes a parsing and serialization step to transform out from and into the stored format in the database.

Customizing the parser is an advanced topic that you can learn more about in the [Extending the Parser](#) section.

First published

March 9, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: How-to Guides”](#)

[Previous Frequently Asked Questions](#) [Previous: Frequently Asked Questions](#)
[Next Accessibility](#) [Next: Accessibility](#)

Accessibility

[↑ Back to top](#)

Accessibility documentation for developers working on the Gutenberg Project.

For more information on accessibility and WordPress see the [Make WordPress Accessibility Handbook](#) and the [Accessibility Team section](#).

Landmark regions

It is a best practice to include ALL content on the page in landmarks, so that screen reader users who rely on them to navigate from section to section do not lose track of content.

For setting up navigation between different regions, see the [navigateRegions package](#) for additional documentation.

Read more regarding landmark design from W3C:

- [General Principles of Landmark Design](#)

- [ARIA Landmarks Examples](#)
- [HTML5 elements that by default define ARIA landmarks](#)

First published

March 9, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Accessibility”](#)

[Previous How-to Guides](#) [Previous: How-to Guides](#)
[Next Blocks](#) [Next: Blocks](#)

Blocks

[↑ Back to top](#)

The purpose of this tutorial is to step through the fundamentals of creating a new block type. Beginning with the simplest possible example, each new section will incrementally build upon the last to include more of the common functionality you could expect to need when implementing your own block types.

To follow along with this tutorial, you can download the [accompanying WordPress plugin](#) which includes all of the examples for you to try on your own site. At each step along the way, experiment by modifying the examples with your own ideas, and observe the effects they have on the block’s behavior.

To find the latest version of the .zip file go to the repo’s [releases page](#) and look in the latest release under ‘Assets’.

Code snippets are provided in two formats “JSX” and “Plain”. JSX refers to JavaScript code that uses JSX syntax which requires a build step. Plain refers to “classic” JavaScript that does not require building. You can change between them using tabs found above each code example. Using JSX, does require you to run [the JavaScript build step](#) to compile your code to a browser compatible format.

Note that it is not required to use JSX to create blocks or extend the editor, you can use classic JavaScript. However, once familiar with JSX and the build step, many developers tend to find it is easier to read and write, thus most code examples you’ll find use the JSX syntax.

First published

March 9, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Blocks”](#)

[Previous Accessibility](#) [Previous: Accessibility](#)

[Next Use styles and stylesheets](#) [Next: Use styles and stylesheets](#)

Use styles and stylesheets

In this article

[Table of Contents](#)

- [Overview](#)
- [Before you start](#)
- [Methods to add style](#)
- [Method 1: Inline style](#)
- [Method 2: Block classname](#)
 - [Build or add dependency](#)
 - [Enqueue stylesheets](#)
- [Conclusion](#)

[↑ Back to top](#)

Overview

A block typically inserts markup (HTML) into post content that you want to style in some way. This guide walks through a few different ways you can use CSS with the block editor and how to work with styles and stylesheets.

Before you start

You will need a basic block and WordPress development environment to implement the examples shown in this guide. See the [Quick Start Guide](#) or [block tutorial](#) to get set up.

Methods to add style

The following are different methods you can use to add style to your block, either in the editor or when saved.

Method 1: Inline style

The first method shows adding the style inline. This transforms the defined style into a property on the element inserted.

The `useBlockProps` React hook is used to set and apply properties on the block’s wrapper element. The following example shows how:

```

import { registerBlockType } from '@wordpress/blocks';
import { useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'gutenberg-examples/example-02-stylesheets', {
    edit() {
        const greenBackground = {
            backgroundColor: '#090',
            color: '#fff',
            padding: '20px',
        };

        const blockProps = useBlockProps( { style: greenBackground } );

        return (
            <p { ...blockProps }>Hello World (from the editor, in green).<
        );
    },
    save() {
        const redBackground = {
            backgroundColor: '#900',
            color: '#fff',
            padding: '20px',
        };

        const blockProps = useBlockProps.save( { style: redBackground } );

        return (
            <p { ...blockProps }>Hello World (from the frontend, in red).<
        );
    },
} );

```

Method 2: Block classname

The inline style works well for a small amount of CSS to apply. If you have much more than the above you will likely find that it is easier to manage with them in a separate stylesheet file.

The `useBlockProps` hook includes the classname for the block automatically, it generates a name for each block using the block's name prefixed with `wp-block-`, replacing the / namespace separator with a single -.

For example the block name: `gutenberg-examples/example-02-stylesheets` would get the classname: `wp-block-gutenberg-examples-example-02-stylesheets`. It might be a bit long but best to avoid conflicts with other blocks.

```

import { registerBlockType } from '@wordpress/blocks';
import { useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'gutenberg-examples/example-02-stylesheets', {
    edit() {
        const blockProps = useBlockProps();

        return (

```

```

        <p { ...blockProps }>Hello World (from the editor, in green).<
    );
},
save() {
    const blockProps = useBlockProps.save();

    return (
        <p { ...blockProps }>Hello World (from the frontend, in red).<
    );
},
);
}
);

```

Build or add dependency

In order to include the blockEditor as a dependency, make sure to run the build step, or update the asset php file.

Build the scripts and update the asset file which is used to keep track of dependencies and the build version.

`npm run build`

Enqueue stylesheets

Like scripts, you can enqueue your block's styles using the `block.json` file.

Use the `editorStyle` property to a CSS file you want to load in the editor view, and use the `style` property for a CSS file you want to load on the frontend when the block is used.

It is worth noting that, if the editor content is iframed, both of these will load in the iframe. `editorStyle` will also load outside the iframe, so it can be used for editor content as well as UI.

For example:

```
{
    "apiVersion": 3,
    "name": "gutenberg-examples/example-02-stylesheets",
    "title": "Example: Stylesheets",
    "icon": "universal-access-alt",
    "category": "layout",
    "editorScript": "file:./block.js",
    "editorStyle": "file:./editor.css",
    "style": "file:./style.css"
}
```

So in your plugin directory, create an `editor.css` file to load in editor view:

```
/* green background */
.wp-block-gutenberg-examples-example-02-stylesheets {
    background: #090;
    color: white;
    padding: 20px;
}
```

And a `style.css` file to load on the frontend:

```
/* red background */
.wp-block-gutenberg-examples-example-02-stylesheets {
    background: #900;
    color: white;
    padding: 20px;
}
```

The files will automatically be enqueued when specified in the `block.json`.

Note: If you have multiple files to include, you can use standard `wp_enqueue_style` functions like any other plugin or theme. You will want to use the following hooks for the block editor:

- `enqueue_block_editor_assets` – to load only in editor view
- `enqueue_block_assets` – loads both on frontend and editor view

[Conclusion](#)

This guide showed a couple of different ways to apply styles to your block, by either inline or in its own style sheet. Both of these methods use the `useBlockProps` hook, see the [block wrapper reference documentation](#) for additional details.

See the complete [stylesheets-79a4c3](#) code in the [block-development-examples repository](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Use styles and stylesheets”](#)

[Previous Blocks](#) [Previous: Blocks](#)

[Next Creating dynamic blocks](#) [Next: Creating dynamic blocks](#)

Creating dynamic blocks

[↑ Back to top](#)

Dynamic blocks are blocks that build their structure and content on the fly when the block is rendered on the front end.

There are two primary uses for dynamic blocks:

1. Blocks where content should change even if a post has not been updated. One example from WordPress itself is the Latest Posts block. This block will update everywhere it is used when a new post is published.
2. Blocks where updates to the code (HTML, CSS, JS) should be immediately shown on the front end of the website. For example, if you update the structure of a block by adding a new class, adding an HTML element, or changing the layout in any other way, using a dynamic block ensures those changes are applied immediately on all occurrences of that block across the site. (If a dynamic block is not used then when block code is updated Gutenberg's [validation process](#) generally applies, causing users to see the validation message, "This block appears to have been modified externally").

For many dynamic blocks, the `save` callback function should be returned as `null`, which tells the editor to save only the [block attributes](#) to the database. These attributes are then passed into the server-side rendering callback, so you can decide how to display the block on the front end of your site. When you return `null`, the editor will skip the block markup validation process, avoiding issues with frequently-changing markup.

If you are using [InnerBlocks](#) in a dynamic block you will need to save the `InnerBlocks` in the `save` callback function using `<InnerBlocks.Content/>`

You can also save an HTML representation of the block. If you provide a server-side rendering callback, this HTML will be replaced with the output of your callback, but will be rendered if your block is deactivated or your render callback is removed.

Block attributes can be used for any content or setting you want to save for that block. In the first example above, with the latest posts block, the number of latest posts you want to show could be saved as an attribute. Or in the second example, attributes can be used for each piece of content you want to show in the front end – such as heading text, paragraph text, an image, a URL, etc.

The following code example shows how to create a dynamic block that shows only the last post as a link.

```
import { registerBlockType } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';
import { useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'gutenberg-examples/example-dynamic', {
    apiVersion: 3,
    title: 'Example: last post',
    icon: 'megaphone',
    category: 'widgets',

    edit: () => {
        const blockProps = useBlockProps();
        const posts = useSelect( ( select ) => {
            return select( 'core' ).getEntityRecords( 'postType', 'post' )
        }, [] );

        return (
            <div { ...blockProps }>
                { ! posts && 'Loading' }
                { posts && posts.length === 0 && 'No Posts' }
                { posts && posts.length > 0 && (
                    <p>{ posts[0].title }</p>
                    <p>{ posts[0].content }</p>
                ) }
            </div>
        );
    },
} );
```

```

                <a href={ posts[ 0 ].link }>
                    { posts[ 0 ].title.rendered }
                </a>
            )
        );
    },
);
}
);

```

Because it is a dynamic block it doesn't need to override the default `save` implementation on the client. Instead, it needs a server component. The contents in the front of your site depend on the function called by the `render_callback` property of `register_block_type`.

```

<?php

/**
 * Plugin Name: Gutenberg examples dynamic
 */

function gutenberg_examples_dynamic_render_callback( $block_attributes, $c
    $recent_posts = wp_get_recent_posts( array(
        'numberposts' => 1,
        'post_status' => 'publish',
    ) );
    if ( count( $recent_posts ) === 0 ) {
        return 'No posts';
    }
    $post = $recent_posts[ 0 ];
    $post_id = $post['ID'];
    return sprintf(
        '<a class="wp-block-my-plugin-latest-post" href="%1$s">%2$s</a>',
        esc_url( get_permalink( $post_id ) ),
        esc_html( get_the_title( $post_id ) )
    );
}

function gutenberg_examples_dynamic() {
    // automatically load dependencies and version
    $asset_file = include( plugin_dir_path( __FILE__ ) . 'build/index.asse
        wp_register_script(
            'gutenberg-examples-dynamic',
            plugins_url( 'build/block.js', __FILE__ ),
            $asset_file['dependencies'],
            $asset_file['version']
        );

        register_block_type( 'gutenberg-examples/example-dynamic', array(
            'api_version' => 3,
            'editor_script' => 'gutenberg-examples-dynamic',
            'render_callback' => 'gutenberg_examples_dynamic_render_callback'
        ) );
    }
}

```

```
add_action( 'init', 'gutenberg_examples_dynamic' );
```

There are a few things to notice:

- The `edit` function still shows a representation of the block in the editor's context (this could be very different from the rendered version, it's up to the block's author)
- The built-in `save` function just returns `null` because the rendering is performed server-side.
- The server-side rendering is a function taking the block and the block inner content as arguments, and returning the markup (quite similar to shortcodes)

Note : For common customization settings including color, border, spacing customization and more, we will see on the [next chapter](#) how you can rely on block supports to provide such functionality in an efficient way.

Live rendering in the block editor

Gutenberg 2.8 added the [`<ServerSideRender>`](#) block which enables rendering to take place on the server using PHP rather than in JavaScript.

Server-side render is meant as a fallback; client-side rendering in JavaScript is always preferred (client rendering is faster and allows better editor manipulation).

```
import { registerBlockType } from '@wordpress/blocks';
import ServerSideRender from '@wordpress/server-side-render';
import { useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'gutenberg-examples/example-dynamic', {
    apiVersion: 3,
    title: 'Example: last post',
    icon: 'megaphone',
    category: 'widgets',

    edit: function ( props ) {
        const blockProps = useBlockProps();
        return (
            <div { ...blockProps }>
                <ServerSideRender
                    block="gutenberg-examples/example-dynamic"
                    attributes={ props.attributes } />
            </div>
        );
    },
} );
```

Note that this code uses the `wp-server-side-render` package but not `wp-data`. Make sure to update the dependencies in the PHP code. You can use `wp-scripts` to automatically build dependencies (see the [block-development-examples repo](#) for PHP code setup).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Creating dynamic blocks”](#)

[Previous Use styles and stylesheets](#) [Previous: Use styles and stylesheets](#)

[Next Nested Blocks: Using InnerBlocks](#) [Next: Nested Blocks: Using InnerBlocks](#)

Nested Blocks: Using InnerBlocks

In this article

Table of Contents

- [Allowed blocks](#)
- [Orientation](#)
- [Default block](#)
- [Template](#)
 - [Post template](#)
- [Using parent, ancestor and children relationships in blocks](#)
 - [Defining parent block relationship](#)
 - [Defining an ancestor block relationship](#)
 - [Defining a children block relationship](#)
- [Using a React hook](#)

[↑ Back to top](#)

You can create a single block that nests other blocks using the [InnerBlocks](#) component. This is used in the Columns block, Social Links block, or any block you want to contain other blocks.

Note: A single block can only contain one [InnerBlocks](#) component.

Here is the basic [InnerBlocks](#) usage.

```
import { registerBlockType } from '@wordpress/blocks';
import { InnerBlocks, useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'gutenberg-examples/example-06', {
    // ...

    edit: () => {
        const blockProps = useBlockProps();

        return (
            <div { ...blockProps }>
                <InnerBlocks />
            </div>
        );
    }
};
```

```

        } ,

      save: () => {
        const blockProps = useBlockProps.save();

        return (
          <div { ...blockProps }>
            <InnerBlocks.Content />
          </div>
        );
      },
    } );
}

```

Allowed blocks

Using the `allowedBlocks` prop, you can further limit, in addition to the `allowedBlocks` field in `block.json`, which blocks can be inserted as direct descendants of this block. It is useful to determine the list of allowed blocks dynamically, individually for each block. For example, determined by a block attribute:

```

const { allowedBlocks } = attributes;
//...
<InnerBlocks allowedBlocks={ allowedBlocks } />;

```

If the list of allowed blocks is always the same, prefer the [allowedBlocks block setting](#) instead.

Orientation

By default, `InnerBlocks` expects its blocks to be shown in a vertical list. A valid use-case is to style inner blocks to appear horizontally, for instance by adding CSS flex or grid properties to the inner blocks wrapper. When blocks are styled in such a way, the `orientation` prop can be set to indicate that a horizontal layout is being used:

```
<InnerBlocks orientation="horizontal" />
```

Specifying this prop does not affect the layout of the inner blocks, but results in the block mover icons in the child blocks being displayed horizontally, and also ensures that drag and drop works correctly.

Default block

By default `InnerBlocks` opens a list of permitted blocks via `allowedBlocks` when the block appender is clicked. You can modify the default block and its attributes that are inserted when the initial block appender is clicked by using the `defaultBlock` property. For example:

```
<InnerBlocks defaultBlock={['core/paragraph', {placeholder: "Lorem ipsum.."}]} />
```

By default this behavior is disabled until the `directInsert` prop is set to `true`. This allows you to specify conditions for when the default block should or should not be inserted.

Template

Use the template property to define a set of blocks that prefill the InnerBlocks component when inserted. You can set attributes on the blocks to define their use. The example below shows a book review template using InnerBlocks component and setting placeholders values to show the block usage.

```
const MY_TEMPLATE = [
  [ 'core/image', {} ],
  [ 'core/heading', { placeholder: 'Book Title' } ],
  [ 'core/paragraph', { placeholder: 'Summary' } ],
];

// ...

edit: () => {
  return (
    <InnerBlocks
      template={ MY_TEMPLATE }
      templateLock="all"
    />
  );
},
```

Use the `templateLock` property to lock down the template. Using `all` locks the template completely so no changes can be made. Using `insert` prevents additional blocks from being inserted, but existing blocks can be reordered. See [templateLock documentation](#) for additional information.

Post template

Unrelated to InnerBlocks but worth mentioning here, you can create a [post template](#) by post type, that preloads the block editor with a set of blocks.

The InnerBlocks template is for the component in the single block that you created, the rest of the post can include any blocks the user likes. Using a post template, can lock the entire post to just the template you define.

```
add_action( 'init', function() {
  $post_type_object = get_post_type_object( 'post' );
  $post_type_object->template = array(
    array( 'core/image' ),
    array( 'core/heading' )
  );
} );
```

Using parent, ancestor and children relationships in blocks

A common pattern for using InnerBlocks is to create a custom block that will only be available if its parent block is inserted. This allows builders to establish a relationship between blocks, while

limiting a nested block's discoverability. There are three relationships that builders can use: `parent`, `ancestor` and `allowedBlocks`. The differences are:

- If you assign a `parent` then you're stating that the nested block can only be used and inserted as a **direct descendant of the parent**.
- If you assign an `ancestor` then you're stating that the nested block can only be used and inserted as a **descendant of the parent**.
- If you assign the `allowedBlocks` then you're stating a relationship in the opposite direction, i.e., which blocks can be used and inserted as **direct descendants of this block**.

The key difference between `parent` and `ancestor` is `parent` has finer specificity, while an `ancestor` has greater flexibility in its nested hierarchy.

Defining parent block relationship

An example of this is the Column block, which is assigned the `parent` block setting. This allows the Column block to only be available as a nested direct descendant in its parent Columns block. Otherwise, the Column block will not be available as an option within the block inserter. See [Column code for reference](#).

When defining a direct descendent block, use the `parent` block setting to define which block is the parent. This prevents the nested block from showing in the inserter outside of the InnerBlock it is defined for.

```
{  
  "title": "Column",  
  "name": "core/column",  
  "parent": [ "core/columns" ],  
  // ...  
}
```

Defining an ancestor block relationship

An example of this is the Comment Author Name block, which is assigned the `ancestor` block setting. This allows the Comment Author Name block to only be available as a nested descendant in its ancestral Comment Template block. Otherwise, the Comment Author Name block will not be available as an option within the block inserter. See [Comment Author Name code for reference](#).

The `ancestor` relationship allows the Comment Author Name block to be anywhere in the hierarchical tree, and not *just* a direct child of the parent Comment Template block, while still limiting its availability within the block inserter to only be visible as an option to insert if the Comment Template block is available.

When defining a descendent block, use the `ancestor` block setting. This prevents the nested block from showing in the inserter outside of the InnerBlock it is defined for.

```
{  
  "title": "Comment Author Name",  
  "name": "core/comment-author-name",  
  "ancestor": [ "core/comment-template" ],  
  // ...  
}
```

Defining a children block relationship

An example of this is the Navigation block, which is assigned the `allowedBlocks` block setting. This makes only a certain subset of block types to be available as direct descendants of the Navigation block. See [Navigation code for reference](#).

The `allowedBlocks` setting can be extended by builders of custom blocks. The custom block can hook into the `blocks.registerBlockType` filter and add itself to the available children of the Navigation.

When defining a set of possible descendant blocks, use the `allowedBlocks` block setting. This limits what blocks are showing in the inserter when inserting a new child block.

```
{
```

```
    "title": "Navigation",
    "name": "core/navigation",
    "allowedBlocks": [ "core/navigation-link", "core/search", "core/social"
        // ...
}
```

Using a React hook

You can use a react hook called `useInnerBlocksProps` instead of the `InnerBlocks` component. This hook allows you to take more control over the markup of inner blocks areas.

The `useInnerBlocksProps` is exported from the `@wordpress/block-editor` package same as the `InnerBlocks` component itself and supports everything the component does. It also works like the `useBlockProps` hook.

Here is the basic `useInnerBlocksProps` hook usage.

```
import { registerBlockType } from '@wordpress/blocks';
import { useBlockProps, useInnerBlocksProps } from '@wordpress/block-editor'

registerBlockType( 'gutenberg-examples/example-06', {
    // ...

    edit: () => {
        const blockProps = useBlockProps();
        const innerBlocksProps = useInnerBlocksProps();

        return (
            <div { ...blockProps }>
                <div {...innerBlocksProps} />
            </div>
        );
    },
    save: () => {
        const blockProps = useBlockProps.save();
        const innerBlocksProps = useInnerBlocksProps.save();

        return (

```

```

        <div { ...blockProps }>
            <div {...innerBlocksProps} />
        </div>
    );
},
);
}
);

```

This hook can also pass objects returned from the `useBlockProps` hook to the `useInnerBlocksProps` hook. This reduces the number of elements we need to create.

```

import { registerBlockType } from '@wordpress/blocks';
import { useBlockProps, useInnerBlocksProps } from '@wordpress/block-editor';

registerBlockType( 'gutenberg-examples/example-06', {
    // ...

    edit: () => {
        const blockProps = useBlockProps();
        const innerBlocksProps = useInnerBlocksProps( blockProps );

        return (
            <div {...innerBlocksProps} />
        );
    },
    save: () => {
        const blockProps = useBlockProps.save();
        const innerBlocksProps = useInnerBlocksProps.save( blockProps );

        return (
            <div {...innerBlocksProps} />
        );
    },
}
);

```

The above code will render to the following markup in the editor:

```
<div>
    <!-- Inner Blocks get inserted here -->
</div>
```

Another benefit to using the hook approach is using the returned value, which is just an object, and deconstruct to get the react children from the object. This property contains the actual child inner blocks thus we can place elements on the same level as our inner blocks.

```

import { registerBlockType } from '@wordpress/blocks';
import { useBlockProps, useInnerBlocksProps } from '@wordpress/block-editor';

registerBlockType( 'gutenberg-examples/example-06', {
    // ...

    edit: () => {
        const blockProps = useBlockProps();
        const { children, ...innerBlocksProps } = useInnerBlocksProps( blo

```

```
        return (
          <div {...innerBlocksProps}>
            { children }
            <!-- Insert any arbitrary html here at the same level as the
                inner blocks -->
          </div>
        );
      },
    // ...
  );
}

<div>
  <!-- Inner Blocks get inserted here -->
  <!-- The custom html gets rendered on the same level -->
</div>
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Nested Blocks: Using InnerBlocks”](#)

[Previous Creating dynamic blocks](#) [Previous: Creating dynamic blocks](#)

[Next Extending the Query Loop block](#) [Next: Extending the Query Loop block](#)

Extending the Query Loop block

In this article

Table of Contents

- [Extending the block with variations](#)
 - [Offer sensible defaults](#)
 - [Customize your variation layout](#)
 - [Making Gutenberg recognize your variation](#)
- [Extending the query](#)
 - [Disabling irrelevant or unsupported query controls](#)
 - [Adding additional controls](#)
 - [Making your custom query work on the front-end side](#)
 - [Making your custom query work on the editor side](#)

[↑ Back to top](#)

The Query Loop block is a powerful tool that allows users to cycle through a determined list of posts and display a certain set of blocks that will inherit the context of each of the posts in the list. For example, it can be set to cycle through all the posts of a certain category and for each of those posts display their featured image. And much more, of course!

But precisely because the Query Loop block is so powerful and allows for great customization, it can also be daunting. Most users wouldn't want to be presented with the full capabilities of the Query Loop block, as most users wouldn't be familiar with the concept of a "query" and its associated technical terms. Instead, most users will likely appreciate a pre-set version of the block, with fewer settings to adjust and clearer naming. The Post List variation offered by default is a good example of this practice: the user will be using the Query Loop block without being exposed to its technicalities, and will also be more likely to discover and understand the purpose of the block.

In the same manner, a lot of extenders might need a way to present bespoke versions of the block, with their own presets, additional settings and without customization options which are irrelevant to their use-case (often, for example, their custom post type). The Query Loop block offers very powerful ways to create such variations.

Extending the block with variations

By registering your own block variation with some specific Query Loop block settings, you can have finer control over how it is presented, while still being able to use the full capabilities which the Query Loop block offers underneath. If you are not familiar with block variations, learn more about them [here](#).

With the block variations API you can provide the default settings that make the most sense for your use-case.

Let's go on a journey, for example, of setting up a variation for a plugin which registers a book [custom post type](#).

Offer sensible defaults

Your first step would be to create a variation which will be set up in such a way to provide a block variation which will display by default a list of books instead of blog posts. The full variation code will look something like this:

```
const MY_VARIATION_NAME = 'my-plugin/books-list';

registerBlockVariation( 'core/query', {
    name: MY_VARIATION_NAME,
    title: 'Books List',
    description: 'Displays a list of books',
    isActive: ( { namespace, query } ) => {
        return (
            namespace === MY_VARIATION_NAME
            && query.postType === 'book'
        );
    },
    icon: /** An SVG icon can go here*/,
    attributes: {
        namespace: MY_VARIATION_NAME,
        query: {

```

```

        perPage: 6,
        pages: 0,
        offset: 0,
        postType: 'book',
        order: 'desc',
        orderBy: 'date',
        author: '',
        search: '',
        exclude: [],
        sticky: '',
        inherit: false,
    },
},
scope: [ 'inserter' ],
}
);

```

If that sounds like a lot, don't fret, let's go through each of the properties here and see why they are there and what they are doing.

Essentially, you would start with something like this:

```

registerBlockVariation( 'core/query', {
    name: 'my-plugin/books-list',
    attributes: {
        query: {
            /** ...more query settings if needed */
            postType: 'book',
        },
    },
} );

```

In this way, the users won't have to choose the custom `postType` from the dropdown, and be already presented with the correct configuration. However, you might ask, how is a user going to find and insert this variation? Good question! To enable this, you should add:

```
{
    /** ...variation properties */
    scope: [ 'inserter' ],
}
```

In this way, your block will show up just like any other block while the user is in the editor and searching for it. At this point you might also want to add a custom icon, title and description to your variation, just like so:

```
{
    /** ...variation properties */
    title: 'Books List',
    description: 'Displays a list of books',
    icon: /* Your svg icon here */,
}
```

At this point, your custom variation will be virtually indistinguishable from a stand-alone block. Completely branded to your plugin, easy to discover and directly available to the user as a drop in.

Customize your variation layout

Please note that the Query Loop block supports 'block' as a string in the scope property. In theory, that's to allow the variation to be picked up after inserting the block itself. Read more about the Block Variation Picker [here](#).

However, it is **unadvisable** to use this currently, this is due to the Query Loop setup with patterns and scope: ['block'] variations, all of the selected pattern's attributes will be used except for postType and inherit query properties, which will likely lead to conflicts and non-functional variations.

To circumvent this, there two routes, the first one is to add your default innerBlocks, like so:

```
innerBlocks: [
  [
    'core/post-template',
    {},
    [ [ 'core/post-title' ], [ 'core/post-excerpt' ] ],
  ],
  [ 'core/query-pagination' ],
  [ 'core/query-no-results' ],
],
```

By having innerBlocks in your variation you essentially skip the setup phase of Query Loop block with suggested patterns and the block is inserted with these inner blocks as its starting content.

The other way would be to register patterns specific to your variation, which are going to be suggested in the setup, and replace flows of the block.

The Query Loop block determines if there is an active variation of itself and if there are specific patterns available for this variation. If there are, these patterns are going to be the only ones suggested to the user, without including the default ones for the original Query Loop block. Otherwise, if there are no such patterns, the default ones are going to be suggested.

In order for a pattern to be “connected” with a Query Loop variation, you should add the name of your variation prefixed with the Query Loop name (e.g. core/query/\$variation_name) to the pattern's blockTypes property. For more details about registering patterns [see here](#).

If you have not provided innerBlocks in your variation, there is also a way to suggest “connected” variations when the user selects Start blank in the setup phase. This is handled in a similar fashion with “connected” patterns, by checking if there is an active variation of Query Loop and if there are any connected variations to suggest.

In order for a variation to be connected to another Query Loop variation we need to define the scope attribute with ['block'] as value and the namespace attribute defined as an array. This array should contain the names(name property) of any variations they want to be connected to.

For example, if we have a Query Loop variation exposed to the inserter(scope : ['inserter']) with the name products, we can connect a scoped block variation by setting its namespace attribute to ['products']. If the user selects this variation after having clicked Start blank, the namespace attribute will be overridden by the main inserter variation.

Making Gutenberg recognize your variation

There is one slight problem you might have realized after implementing this variation: while it is transparent to the user as they are inserting it, Gutenberg will still recognize the variation as a Query Loop block at its core and so, after its insertion, it will show up as a Query Loop block in the tree view of the editor, for instance.

We need a way to tell the editor that this block is indeed your specific variation. This is what the `isActive` property is made for: it's a way to determine whether a certain variation is active based on the block's attributes. You could use it like this:

```
{  
    /** ...variation properties */  
    isActive: ( { namespace, query } ) => {  
        return (  
            namespace === MY_VARIATION_NAME  
            && query.postType === 'book'  
        );  
    },  
}
```

You might be tempted to only compare the `postType` so that Gutenberg will recognize the block as your variation any time the `postType` matches `book`. This casts a net too wide, however, as other plugins might want to publish variations based on the `book` post type too, or we might just not want the variation to be recognized every time the user sets the type to `book` manually through the editor settings.

That's why the Query Loop block exposes a special attribute called `namespace`. It really doesn't do anything inside the block implementation, and it's used as an easy and consistent way for extenders to recognize and scope their own variation. In addition, `isActive` also accepts just an array of strings with the attributes to compare. Often, `namespace` would be sufficient, so you would use it like so:

```
{  
    /** ...variation properties */  
    attributes: {  
        /** ...variation attributes */  
        namespace: 'my-plugin/books-list',  
    },  
    isActive: [ 'namespace' ],  
}
```

Like so, Gutenberg will know that it is your specific variation only in the case it matches your custom namespace! So convenient!

Extending the query

Even with all of this, your custom post type might have unique requirements: it might support certain custom attributes that you might want to filter and query for, or some other query parameters might be irrelevant or even completely unsupported! We have build the Query Loop block with such use-cases in mind, so let's see how you can solve this problem.

Disabling irrelevant or unsupported query controls

Let's say you don't use at all the `sticky` attribute in your books, so that would be totally irrelevant to the customization of your block. In order to not confuse the users as to what a setting might do, and only exposing a clear UX to them, we want this control to be unavailable. Furthermore, let's say that you don't use the `author` field at all, which generally indicates the person who has added that post to the database, instead you use a custom `bookAuthor` field. As such, not only keeping the `author` filter would be confusing, it would outright "break" your query.

For this reason, the Query Loop block variations support a property called `allowedControls`, which accepts an array of keys of the controls we want to display on the inspector sidebar. By default, we accept all the controls, but as soon as we provide an array to this property, we want to specify only the controls which are going to be relevant for us!

As of Gutenberg version 14.2, the following controls are available:

- `inherit` – Shows the toggle switch for allowing the query to be inherited directly from the template.
- `postType` – Shows a dropdown of available post types.
- `order` – Shows a dropdown to select the order of the query.
- `sticky` – Shows a dropdown to select how to handle sticky posts.
- `taxQuery` – Shows available taxonomies filters for the currently selected post type.
- `author` – Shows an input field to filter the query by author.
- `search` – Shows an input filed to filter the query by keywords.

In our case, the property would look like this:

```
{  
    /** ...variation properties */  
    allowedControls: [ 'inherit', 'order', 'taxQuery', 'search' ],  
}
```

If you want to hide all the above available controls, you can set an empty array as a value of `allowedControls`.

Notice that we have also disabled the `postType` control. When the user selects our variation, why show them a confusing dropdown to change the post type? On top of that it might break the block as we can implement custom controls, as we'll see shortly.

Adding additional controls

Because our plugin uses custom attributes that we need to query, we want to add our own controls to allow the users to select those instead of the ones we have just disabled from the core inspector controls. We can do this via a [React HOC](#) hooked into a [block filter](#), like so:

```
import { InspectorControls } from '@wordpress/block-editor';  
  
export const withBookQueryControls = ( BlockEdit ) => ( props ) => {  
    // We only want to add these controls if it is our variation,  
    // so here we can implement a custom logic to check for that, similar  
    // to the `isActive` function described above.  
    // The following assumes that you wrote a custom `isMyBooksVariation`  
    // function to handle that.
```

```

        return isMyBooksVariation( props ) ? (
            <>
                <BlockEdit key="edit" { ...props } />
                <InspectorControls>
                    <BookAuthorSelector /> { /** Our custom component */ }
                </InspectorControls>
            </>
        ) : (
            <BlockEdit key="edit" { ...props } />
        );
    };
}

addFilter( 'editor.BlockEdit', 'core/query', withBookQueryControls );

```

Of course, you'll be responsible for implementing the logic of your control (you might want to take a look at [@wordpress/components](#) to make your controls fit seamlessly within the Gutenberg UI). Any extra parameter you assign within the `query` object inside the blocks attributes can be used to create a custom query according to your needs, with a little extra effort.

Currently, you'll likely have to implement slightly different paths to make the query behave correctly both on the front-end side (i.e. on the end user's side) and to show the correct preview on the editor side.

```

{
    /** ...variation properties */
    attributes: {
        /** ...variation attributes */
        query: {
            /** ...more query settings if needed */
            postType: 'book',
            /** Our custom query parameter */
            bookAuthor: 'J. R. R. Tolkien'
        }
    }
}

```

Making your custom query work on the front-end side

The Query Loop block functions mainly through the Post Template block which receives the attributes and builds the query from there. Other first-class children of the Query Loop block (such as the Pagination block) behave in the same way. They build their query and then expose the result via the filter [query_loop_block_query_vars](#).

You can hook into that filter and modify your query accordingly. Just make sure you don't cause side-effects to other Query Loop blocks by at least checking that you apply the filter only to your variation!

```

if( 'my-plugin/books-list' === $block[ 'attrs' ][ 'namespace' ] ) {
    add_filter(
        'query_loop_block_query_vars',
        function( $query ) {
            /** You can read your block custom query parameters here and b
        },

```

```
) ;  
}
```

(In the code above, we assume you have some way to access the block, for example within a [`pre_render_block`](#) filter, but the specific solution can be different depending on the use-case, so this is not a firm recommendation).

[**Making your custom query work on the editor side**](#)

To finish up our custom variation, we might want the editor to react to changes in our custom query and display an appropriate preview accordingly. This is not required for a functioning block, but it enables a fully integrated user experience for the consumers of your block.

The Query Loop block fetches its posts to show the preview using the [WordPress REST API](#). Any extra parameter added to the `query` object will be passed as a query argument to the API. This means that these extra parameters should be either supported by the REST API, or be handled by custom filters such as the [`rest_{\$this->post_type}_query`](#) filter which allows you to hook into any API request for your custom post type. Like so:

```
add_filter(  
    'rest_book_query',  
    function( $args, $request ) {  
        /** We can access our custom parameters from here */  
        $book_author = $request->get_param( 'bookAuthor' );  
        /** ...your custom query logic */  
    }  
);
```

And, just like that, you'll have created a fully functional variation of the Query Loop block!

First published

October 6, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Extending the Query Loop block”](#)

[Previous Nested Blocks: Using InnerBlocks](#) [Previous: Nested Blocks: Using InnerBlocks](#)
[Next Development Platform](#) [Next: Development Platform](#)

Development Platform

In this article

Table of Contents

- [UI components](#)
- [Development scripts](#)

- [Block Editor](#)

[↑ Back to top](#)

The Gutenberg Project is not only building a better editor for WordPress, but also creating a platform to build upon. This platform consists of a set of JavaScript packages and tools that you can use in your web application. [View the list of packages available on npm.](#)

UI components

The [WordPress Components package](#) contains a set of UI components you can use in your project. See the [WordPress Storybook site](#) for an interactive guide to the available components and settings.

Here is a quick example, how to use components in your project.

Install the dependency:

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

Usage in React:

```
import { Button } from '@wordpress/components';

function MyApp() {
    return <Button>Hello Button</Button>;
}
```

Many components include CSS to add style, you will need to include for the components to appear correctly. The component stylesheet can be found in `node_modules/@wordpress/components/build-style/style.css`, you can link directly or copy and include it in your project.

Development scripts

The [@wordpress/scripts package](#) is a collection of reusable scripts for JavaScript development — includes scripts for building, linting, and testing — all with no additional configuration files.

Here is a quick example, on how to use `wp-scripts` tool in your project.

Install the dependency:

```
npm install --save-dev @wordpress/scripts
```

You can then add a scripts section to your `package.json` file, for example:

```
"scripts": {
    "build": "wp-scripts build",
    "format": "wp-scripts format",
    "lint:js": "wp-scripts lint-js",
    "start": "wp-scripts start"
}
```

You can then use `npm run build` to build your project with all the default webpack settings already configured, likewise for formatting and linting. The `start` command is used for development mode. See the [@wordpress/scripts package](#) for full documentation.

For more info, see the [Getting Started with JavaScript tutorial](#) in the Block Editor Handbook.

[Block Editor](#)

The [@wordpress/block-editor package](#) allows you to create and use standalone block editors.

You can learn more by reading the [tutorial “Building a custom block editor”](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Development Platform”](#)

[Previous Extending the Query Loop block](#) [Previous: Extending the Query Loop block](#)
[Next Building a custom block editor](#) [Next: Building a custom block editor](#)

Building a custom block editor

In this article

Table of Contents

- [Introduction](#)
- [Code syntax](#)
- [What you’re going to be building](#)
- [Plugin setup and organization](#)
- [The “Core” of the editor](#)
- [Creating the custom “Block Editor” page](#)
 - [Registering the page](#)
 - [Adding the target HTML](#)
 - [Enqueuing JavaScript and CSS](#)
 - [Inlining the editor settings](#)
- [Registering and rendering the custom block editor](#)
- [Reviewing the <Editor> component](#)
 - [Dependencies](#)
 - [Editor render](#)
 - [Keyboard navigation](#)
- [The custom <BlockEditor>](#)
 - [Understanding the <BlockEditorProvider> component](#)
 - [Understanding the <BlockList> component](#)

- [Reviewing the sidebar](#)
- [Block Persistence](#)
 - [Storing blocks in state](#)
 - [Saving block data](#)
 - [Retrieving previous block data](#)
- [Wrapping up](#)

[↑ Back to top](#)

The WordPress block editor is a powerful tool that allows you to create and format content in various ways. It is powered, in part, by the [@wordpress/block-editor](#) package, which is a JavaScript library that provides the core functionality of the editor.

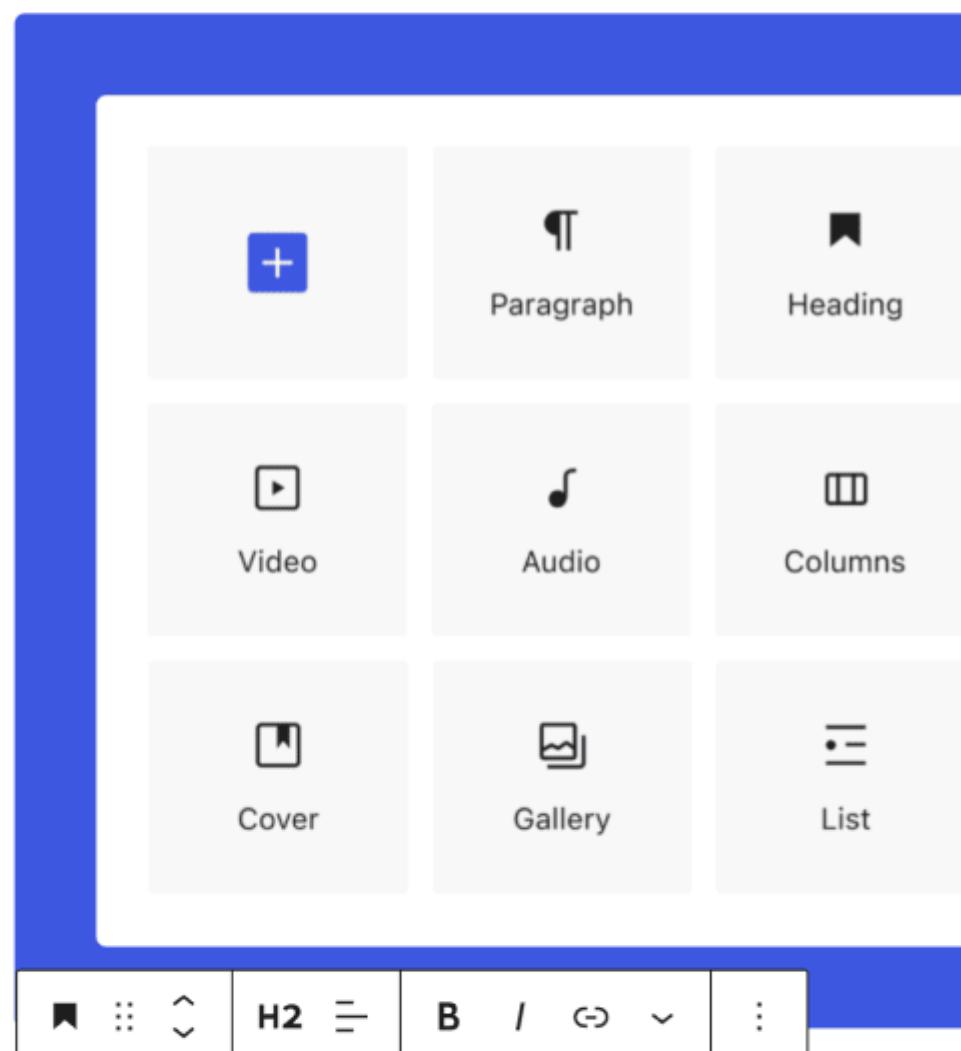
This package can also be used to create custom block editors for virtually any other web application. This means that you can use the same blocks and block editing experience outside of WordPress.

[Dashboard](#)[Posts](#)[Media](#)[Pages](#)[Comments](#)[Appearance](#)[Plugins](#)[Users](#)[Tools](#)[Settings](#)[Block Editor](#)[Collapse menu](#)

Custom Block Editor

The building blocks

Create engaging, interactive, and transferrable content standard, and a collection of design elements with a the blocks beyond WordPress with the flexibility of custom



Creativity without limits

This flexibility and interoperability makes blocks a powerful tool for building and managing content across multiple applications. It also makes it simpler for developers to create content editors that work best for their users.

This guide covers the basics of creating your first custom block editor.

Introduction

With its many packages and components, the Gutenberg codebase can be daunting at first. But at its core, it's all about managing and editing blocks. So if you want to work on the editor, it's essential to understand how block editing works at a fundamental level.

This guide will walk you through building a fully functioning, custom block editor “instance” within WordPress. Along the way, we'll introduce you to the key packages and components, so you can see how the block editor works under the hood.

By the end of this article, you will have a solid understanding of the block editor's inner workings and be well on your way to creating your own block editor instances.

The code used throughout this guide is available for download in the [accompanying WordPress plugin](#). The demo code in this plugin as an essential resource.

Code syntax

The code snippets in this guide use JSX syntax. However, you could use plain JavaScript if you prefer. However, once familiar with JSX, many developers find it easier to read and write, so most code examples in the Block Editor Handbook use this syntax.

What you're going to be building

Throughout this guide, you will create an (almost) fully functioning block editor instance. The result will look something like this:

[Dashboard](#)[Posts](#)[Media](#)[Pages](#)[Comments](#)[Appearance](#)[Plugins](#)[Users](#)[Tools](#)[Settings](#)[Block Editor](#)[Collapse menu](#)

Custom Block Editor

The building blocks

Create engaging, interactive, and transferrable content standard, and a collection of design elements with a the blocks beyond WordPress with the flexibility of custom

A screenshot of the WordPress Custom Block Editor interface. On the left, a sidebar lists various site navigation items like Dashboard, Posts, Media, etc. Below the sidebar, a blue header bar indicates the current section: "Block Editor". The main content area features a large title "The building blocks" followed by a descriptive paragraph. Below this, a prominent blue button labeled "New" is visible. A large, semi-transparent blue box covers the central area, containing a grid of eight block types, each represented by an icon and a label: "Paragraph" (with a text icon), "Heading" (with a bookmark icon), "Video" (with a play icon), "Audio" (with a music note icon), "Columns" (with a grid icon), "Cover" (with a square icon), "Gallery" (with a photo icon), and "List" (with a list icon). At the bottom of the blue box is a toolbar with icons for bold, italic, underline, and other editing functions.

Creativity without limits

While it looks similar, this editor will not be the same *Block Editor* you are familiar with when creating posts and pages in WordPress. Instead, it will be an entirely custom instance that will live within a custom WordPress admin page called “Block Editor.”

The editor will have the following features:

- Ability to add and edit all Core blocks.
- Familiar visual styles and main/sidebar layout.
- *Basic* block persistence between page reloads.

Plugin setup and organization

The custom editor is going to be built as a WordPress plugin. To keep things simple, the plugin will be named **Standalone Block Editor Demo** because that is what it does.

The plugin file structure will look like this:

 src
 .editorconfig
 .eslintrc
 .gitignore
 README.md
 init.php
 package-lock.json
 package.json
 plugin.php
 webpack.config.js

Here is a brief summary of what's going on:

- `plugin.php` – Standard plugin “entry” file with comment meta data, which requires `init.php`.
- `init.php` – Handles the initialization of the main plugin logic.
- `src/` (directory) – This is where the JavaScript and CSS source files will live. These files are *not* directly enqueued by the plugin.
- `webpack.config.js` – A custom Webpack config extending the defaults provided by the [`@wordpress/scripts`](#) npm package to allow for custom CSS styles (via Sass).

The only item not shown above is the `build/` directory, which is where the *compiled* JS and CSS files are outputted by [`@wordpress/scripts`](#). These files are enqueued by the plugin separately.

Throughout this guide, filename references will be placed in a comment at the top of each code snippet so you can follow along.

With the basic file structure in place, let's look at what packages will be needed.

The “Core” of the editor

While the WordPress Editor is comprised of many moving parts, at its core is the [`@wordpress/block-editor`](#) package, which is best summarized by its own README file:

This module allows you to create and use standalone block editors.

Perfect, this is the main package you will use to create the custom block editor instance. But first, you need to create a home for the editor.

Creating the custom “Block Editor” page

Let's begin by creating a custom page within WordPress admin that will house the custom block editor instance.

If you're already comfortable with the process of creating custom admin pages in WordPress, you might want to [skip ahead](#).

Registering the page

To do this, you need to [register a custom admin page](#) using the standard WordPress [`add_menu_page\(\)`](#) helper:

```
// File: init.php

add_menu_page(
    'Standalone Block Editor',           // Visible page name
    'Block Editor',                     // Menu label
    'edit_posts',                      // Required capability
    'getdavesbe',                      // Hook/slug of page
    'getdave_sbe_render_block_editor', // Function to render the page
    'dashicons-welcome-widgets-menus'   // Custom icon
);
```

The `getdave_sbe_render_block_editor` function will be used to render the contents of the admin page. As a reminder, the source code for each step is available in the [accompanying plugin](#).

Adding the target HTML

Since the block editor is a React-powered application, you need to output some HTML into the custom page where JavaScript can render the block editor.

Let's use the `getdave_sbe_render_block_editor` function referenced in the step above.

```
// File: init.php

function getdave_sbe_render_block_editor() {
```

```

?>
<div
    id="getdave-sbe-block-editor"
    class="getdave-sbe-block-editor"
>
    Loading Editor...
</div>
<?php
}

```

The function outputs some basic placeholder HTML. Note the `id` attribute `getdave-sbe-block-editor`, which will be used shortly.

[Enqueuing JavaScript and CSS](#)

With the target HTML in place, you can now enqueue some JavaScript and CSS so that they will run on the custom admin page.

To do this, let's hook into [`admin_enqueue_scripts`](#).

First, you must ensure the custom code is only run on the custom admin page. So, at the top of the callback function, exit early if the page doesn't match the page's identifier:

```

// File: init.php

function getdave_sbe_block_editor_init( $hook ) {

    // Exit if not the correct page.
    if ( 'toplevel_page_getdavesbe' !== $hook ) {
        return;
    }
}

add_action( 'admin_enqueue_scripts', 'getdave_sbe_block_editor_init' );

```

With this in place, you can then safely register the main JavaScript file using the standard WordPress [`wp_enqueue_script\(\)`](#) function:

```

// File: init.php

wp_enqueue_script( $script_handle, $script_url, $script_asset['dependencies'] );

```

To save time and space, the `$script_` variables assignment has been omitted. You can [review these here](#).

Note the third argument for script dependencies, `$script_asset['dependencies']`. These dependencies are dynamically generated using [@wordpress/dependency-extraction-webpack-plugin](#) which will [ensure that](#) WordPress provided scripts are not included in the built bundle.

You also need to register both your custom CSS styles and the WordPress default formatting library to take advantage of some nice default styling:

```
// File: init.php

// Enqueue default editor styles.
wp_enqueue_style( 'wp-format-library' );

// Enqueue custom styles.
wp_enqueue_style(
    'getdave-sbe-styles', // Handle
    plugins_url( 'build/index.css', __FILE__ ), // Block editor CSS
    array( 'wp-edit-blocks' ), // Dependency to include this file
    filemtime( __DIR__ . '/build/index.css' )
);


```

Inlining the editor settings

Looking at the `@wordpress/block-editor` package, you can see that it accepts a [settings object](#) to configure the default settings for the editor. These are available on the server side, so you need to expose them for use within JavaScript.

To do this, let's [inline the settings object as JSON](#) assigned to the global `window.getdaveSbeSettings` object:

```
// File: init.php

// Get custom editor settings.
$settings = getdave_sbe_get_block_editor_settings();

// Inline all settings.
wp_add_inline_script( $script_handle, 'window.getdaveSbeSettings = ' . wp_
```

Registering and rendering the custom block editor

With the PHP above in place to create the admin page, you're now finally ready to use JavaScript to render a block editor into the page's HTML.

Begin by opening the main `src/index.js` file. Then pull in the required JavaScript packages and import the CSS styles. Note that using Sass requires [extending](#) the default `@wordpress/scripts` Webpack config.

```
// File: src/index.js

// External dependencies.
import { createRoot } from 'react-dom';

// WordPress dependencies.
import domReady from '@wordpress/dom-ready';
import { registerCoreBlocks } from '@wordpress/block-library';

// Internal dependencies.
import Editor from './editor';
import './styles.scss';
```

Next, once the DOM is ready you will need to run a function which:

- Grabs the editor settings from `window.getdaveSbeSettings` (previously inlined from PHP).
- Registers all the Core WordPress blocks using `registerCoreBlocks`.
- Renders an `<Editor>` component into the waiting `<div>` on the custom admin page.

```
domReady( function () {
    const root = createRoot( document.getElementById( 'getdave-sbe-block-e' );
    const settings = window.getdaveSbeSettings || {};
    registerCoreBlocks();
    root.render(
        <Editor settings={ settings } />
    );
} );
```

It is possible to render the editor from PHP without creating an unnecessary JS global. Check out the [Edit Site](#) package in the Gutenberg plugin for an example of this.

Reviewing the `<Editor>` component

Let's take a closer look at the `<Editor>` component that was used in the code above and lives in `src/editor.js` of the [companion plugin](#).

Despite its name, this is not the actual core of the block editor. Rather, it is a *wrapper* component that will contain the components that form the custom editor's main body.

Dependencies

The first thing to do inside `<Editor>` is to pull in some dependencies.

```
// File: src/editor.js

import Notices from 'components/notices';
import Header from 'components/header';
import Sidebar from 'components/sidebar';
import BlockEditor from 'components/block-editor';
```

The most important of these are the internal components `BlockEditor` and `Sidebar`, which will be covered shortly.

The remaining components consist mostly of static elements that form the editor's layout and surrounding user interface (UI). These elements include the header and notice areas, among others.

Editor render

With these components available, you can define the `<Editor>` component.

```
// File: src/editor.js

function Editor( { settings } ) {
    return (
```

```

        <DropZoneProvider>
          <div className="getdavesbe-block-editor-layout">
            <Notices />
            <Header />
            <Sidebar />
            <BlockEditor settings={ settings } />
          </div>
        </DropZoneProvider>
      );
    }
  
```

In this process, the core of the editor's layout is being scaffolded, along with a few specialized [context providers](#) that make specific functionality available throughout the component hierarchy.

Let's examine these in more detail:

- <DropZoneProvider> – Enables the use of [dropzones for drag and drop functionality](#)
- <Notices> – Provides a “snack bar” Notice that will be rendered if any messages are dispatched to the `core/notices` store
- <Header> – Renders the static title “Standalone Block Editor” at the top of the editor UI
- <BlockEditor> – The custom block editor component

[**Keyboard navigation**](#)

With this basic component structure in place, the only remaining thing left to do is wrap everything in the [navigateRegions HOC](#) to provide keyboard navigation between the different “regions” in the layout.

```
// File: src/editor.js

export default navigateRegions( Editor );
```

[**The custom <BlockEditor>**](#)

Now the core layouts and components are in place. It's time to explore the custom implementation of the block editor itself.

The component for this is called <BlockEditor>, and this is where the magic happens.

Opening `src/components/block-editor/index.js` reveals that it's the most complex component encountered thus far. A lot going on, so start by focusing on what is being rendered by the <BlockEditor> component:

```
// File: src/components/block-editor/index.js

return (
  <div className="getdavesbe-block-editor">
    <BlockEditorProvider
      value={ blocks }
      onInput={ updateBlocks }
      onChange={ persistBlocks }
      settings={ settings }
    >
      <Sidebar.InspectorFill>
```

```

        <BlockInspector />
      </Sidebar.InspectorFill>
      <BlockCanvas height="400px" />
    </BlockEditorProvider>
  </div>
);

```

The key components are `<BlockEditorProvider>` and `<BlockList>`. Let's examine these.

[Understanding the `<BlockEditorProvider>` component](#)

`<BlockEditorProvider>` is one of the most important components in the hierarchy. It establishes a new block editing context for a new block editor.

As a result, it is *fundamental* to the entire goal of this project.

The children of `<BlockEditorProvider>` comprise the UI for the block editor. These components then have access to data (via `Context`), enabling them to *render* and *manage* the blocks and their behaviors within the editor.

```
// File: src/components/block-editor/index.js
```

```

<BlockEditorProvider
  value={ blocks }          // Array of block objects
  onInput={ updateBlocks }   // Handler to manage Block updates
  onChange={ persistBlocks } // Handler to manage Block updates/persist
  settings={ settings }     // Editor "settings" object
/>

```

BlockEditor props

You can see that `<BlockEditorProvider>` accepts an array of (parsed) block objects as its `value` prop and, when there's a change detected within the editor, calls the `onChange` and/or `onInput` handler prop (passing the new Blocks as an argument).

Internally it does this by subscribing to the provided `registry` (via the [`withRegistryProvider HOC`](#)), listening to block change events, determining whether the block changing was persistent, and then calling the appropriate `onChange` | `Input` handler accordingly.

For the purposes of this simple project, these features allow you to:

- Store the array of current blocks in state as `blocks`.
- Update the `blocks` state in memory on `onInput` by calling the hook setter `updateBlocks(blocks)`.
- Handle basic persistence of blocks into `localStorage` using `onChange`. This is [fired when block updates are considered “committed”](#).

It's also worth recalling that the component accepts a `settings` property. This is where you will add the editor settings inlined earlier as JSON within `init.php`. You can use these settings to configure features such as custom colors, available image sizes, and [much more](#).

Understanding the <BlockList> component

Alongside `<BlockEditorProvider>` the next most interesting component is `<BlockList>`.

This is one of the most important components as it's role is to **render a list of blocks into the editor**.

It does this in part thanks to being placed as a child of `<BlockEditorProvider>`, which affords it full access to all information about the state of the current blocks in the editor.

How does BlockList work?

Under the hood, `<BlockList>` relies on several other lower-level components in order to render the list of blocks.

The hierarchy of these components can be *approximated* as follows:

```
// Pseudo code for example purposes only.
```

```
<BlockList>
  /* renders a list of blocks from the rootClientId. */
  <BlockListBlock>
    /* renders a single block from the BlockList. */
    <BlockEdit>
      /* renders the standard editable area of a block. */
      <Component /> /* renders the block UI as defined by its `edit()`
      */
    </BlockEdit>
  </BlockListBlock>
</BlockList>
```

Here is roughly how this works together to render the list of blocks:

- `<BlockList>` loops over all the block `clientIds` and renders each via `<BlockListBlock />`.
- `<BlockListBlock />`, in turn, renders the individual block using its own subcomponent `<BlockEdit>`.
- Finally, the `block itself` is rendered using the `Component` placeholder component.

The `@wordpress/block-editor` package components are among the most complex and involved. Understanding them is crucial if you want to grasp how the editor functions at a fundamental level. Studying these components is strongly advised.

Reviewing the sidebar

Also within the render of the `<BlockEditor>`, is the `<Sidebar>` component.

```
// File: src/components/block-editor/index.js

return (
  <div className="getdavesbe-block-editor">
    <BlockEditorProvider>
```

```

        <Sidebar.InspectorFill> /* <-- SIDEBAR */
            <BlockInspector />
        </Sidebar.InspectorFill>
        <BlockCanvas height="400px" />
    </BlockEditorProvider>
</div>
);

```

This is used, in part, to display advanced block settings via the `<BlockInspector>` component.

```

<Sidebar.InspectorFill>
    <BlockInspector />
</Sidebar.InspectorFill>

```

However, the keen-eyed readers amongst you will have already noted the presence of a `<Sidebar>` component within the `<Editor>` (`src/editor.js`) component's layout:

```

// File: src/editor.js
<Notices />
<Header />
<Sidebar /> // <-- What's this?
<BlockEditor settings={ settings } />

```

Opening the `src/components/sidebar/index.js` file, you can see that this is, in fact, the component rendered within `<Editor>` above. However, the implementation utilises Slot/Fill to expose a Fill (`<Sidebar.InspectorFill>`), which is subsequently imported and rendered inside of the `<BlockEditor>` component (see above).

With this in place, you then can render `<BlockInspector />` as a child of the `Sidebar.InspectorFill`. This has the result of allowing you to keep `<BlockInspector>` within the React context of `<BlockEditorProvider>` whilst allowing it to be rendered into the DOM in a separate location (i.e. in the `<Sidebar>`).

This might seem overly complex, but it is required in order for `<BlockInspector>` to have access to information about the current block. Without Slot/Fill, this setup would be extremely difficult to achieve.

And with that you have covered the render of your custom `<BlockEditor>`.

[<BlockInspector>](#)
 itself actually renders a Slot for [<InspectorControls>](#). This is what allows you [render](#) a `<InspectorControls>>` component inside the `edit()` definition for your block and have it display within the editor's sidebar. Exploring this component in more detail is recommended.

[Block Persistence](#)

You have come a long way on your journey to create a custom block editor. But there is one major area left to touch upon – block persistence. In other words, having your blocks saved and available *between* page refreshes.

[Dashboard](#)[Posts](#)[Media](#)[Pages](#)[Comments](#)[Appearance](#)[Plugins](#)[Users](#)[Tools](#)[Settings](#)[Block Editor](#)[Collapse menu](#)

Custom Block Editor

The building blocks

Create engaging, interactive, and transferrable content standard, and a collection of design elements with a tool blocks beyond WordPress with the flexibility of custom



Paragraph



Heading



Video



Audio



Columns



Cover



Gallery



List



Creativity without limits

As this is only an *experiment*, this guide has opted to utilize the browser's `localStorage` API to handle saving block data. In a real-world scenario, you would likely choose a more reliable and robust system (e.g. a database).

That said, let's take a closer look at how to handle save blocks.

[Storing blocks in state](#)

Looking at the `src/components/block-editor/index.js` file, you will notice that some state has been created to store the blocks as an array:

```
// File: src/components/block-editor/index.js

const [ blocks, updateBlocks ] = useState( [] );
```

As mentioned earlier, `blocks` is passed to the “controlled” component `<BlockEditorProvider>` as its `value` prop. This “hydrates” it with an initial set of blocks. Similarly, the `updateBlocks` setter is hooked up to the `onInput` callback on `<BlockEditorProvider>`, which ensures that the block state is kept in sync with changes made to blocks within the editor.

[Saving block data](#)

If you now turn your attention to the `onChange` handler, you will notice it is hooked up to a function `persistBlocks()` which is defined as follows:

```
// File: src/components/block-editor/index.js

function persistBlocks( newBlocks ) {
    updateBlocks( newBlocks );
    window.localStorage.setItem( 'getdavesbeBlocks', serialize( newBlocks )
}
```

This function accepts an array of “committed” block changes and calls the state setter `updateBlocks`. It also stores the blocks within LocalStorage under the key `getdavesbeBlocks`. In order to achieve this, the block data is serialized into [Gutenberg “Block Grammar”](#) format, meaning it can be safely stored as a string.

If you open DeveloperTools and inspect the LocalStorage you will see serialized block data stored and updated as changes occur within the editor. Below is an example of the format:

```
<!-- wp:heading -->
<h2>An experiment with a standalone Block Editor in the WordPress admin</h2>
<!-- /wp:heading -->

<!-- wp:paragraph -->
<p>This is an experiment to discover how easy (or otherwise) it is to crea
<!-- /wp:paragraph -->
```

[Retrieving previous block data](#)

Having persistence in place is all well and good, but it's only useful if that data is retrieved and *restored* within the editor upon each full page reload.

Accessing data is a side effect, so you must use the `useEffect` hook to handle this.

```
// File: src/components/block-editor/index.js

useEffect( () => {
  const storedBlocks = window.localStorage.getItem( 'getdavesbeBlocks' )

  if ( storedBlocks && storedBlocks.length ) {
    updateBlocks( () => parse( storedBlocks ) );
    createInfoNotice( 'Blocks loaded', {
      type: 'snackbar',
      isDismissible: true,
    } );
  }
}, [] );
```

This handler:

- Grabs the serialized block data from local storage.
- Converts the serialized blocks back to JavaScript objects using the `parse()` utility.
- Calls the state setter `updateBlocks` causing the `blocks` value to be updated in state to reflect the blocks retrieved from LocalStorage.

As a result of these operations, the controlled `<BlockEditorProvider>` component is updated with the blocks restored from LocalStorage, causing the editor to show these blocks.

Finally, you will want to generate a notice – which will display in the `<Notice>` component as a “snackbar” notice – to indicate that the blocks have been restored.

Wrapping up

Congratulations for completing this guide. You should now have a better understanding of how the block editor works under the hood.

The full code for the custom block editor you have just built is [available on GitHub](#). Download and try it out for yourself. Experiment, then and take things even further.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Building a custom block editor”](#)

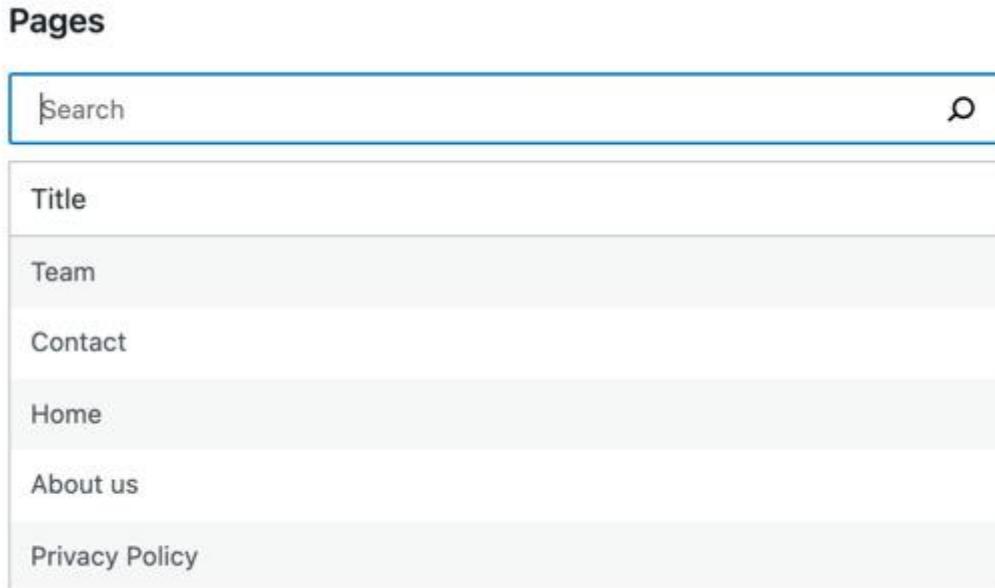
[Previous Development Platform](#) [Previous: Development Platform](#)

[Next Create your First App with Gutenberg Data](#) [Next: Create your First App with Gutenberg Data](#)

Create your First App with Gutenberg Data

[↑ Back to top](#)

This tutorial aims to get you comfortable with the Gutenberg data layer. It guides you through building a simple React application that enables the user to manage their WordPress pages. The finished app will look like this:



You may review the [finished app](#) in the block-development-examples repository.

Table of Contents

1. [Setup](#)
2. [Building a basic list of pages](#)
3. [Building an edit form](#)
4. [Building a *create page* form](#)
5. [Adding a delete button](#)

First published

February 9, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Create your First App with Gutenberg Data”](#)

[Previous Building a custom block editor](#) [Previous: Building a custom block editor](#)

[Next](#) [Setup](#) [Next: Setup](#)

Setup

In this article

Table of Contents

- [Creating a plugin](#)
- [Setting up the build pipeline](#)
- [Testing if it worked](#)
- [What's next?](#)

[↑ Back to top](#)

We will build the application as a WordPress plugin, which means you need to have WordPress itself installed. One way to do this is by following the instructions on the [Getting Started](#) page. Once your setup is complete, you can follow along with the rest of this tutorial.

Also, this tutorial will lean heavily on Redux concepts such as state, actions, and selectors. If you are not familiar with them, you may want to start by reviewing [Getting Started With Redux](#).

[Creating a plugin](#)

We'll do all the development inside of a WordPress plugin. Let's start by creating a `wp-content/plugins/my-first-gutenberg-app` directory in your local WordPress environment. We will need to create four files inside that directory:

- `my-first-gutenberg-app.php` – to create a new admin page
- `src/index.js` – for our JavaScript application
- `style.css` – for the minimal stylesheet
- `package.json` – for the build process

Go ahead and create these files using the following snippets:

src/index.js:

```
import { createRoot } from 'react-dom';

function MyFirstApp() {
    return <span>Hello from JavaScript!</span>;
}

const root = createRoot( document.getElementById( 'my-first-gutenberg-app' ) );
window.addEventListener(
    'load',
    function () {
        root.render(
            <MyFirstApp />,
        );
    }
);
```

```
},
false
);
```

style.css:

```
.toplevel_page_my-first-gutenberg-app #wpcontent {
    background: #fff;
    height: 1000px;
}
button .components-spinner {
    width: 15px;
    height: 15px;
    margin-top: 0;
    margin-bottom: 0;
    margin-left: 0;
}
.form-buttons {
    display: flex;
}
.my-gutenberg-form .form-buttons {
    margin-top: 20px;
    margin-left: 1px;
}
.form-error {
    color: #cc1818;
}
.form-buttons button {
    margin-right: 4px;
}
.form-buttons .components-spinner {
    margin-top: 0;
}
#my-first-gutenberg-app {
    max-width: 500px;
}
#my-first-gutenberg-app ul,
#my-first-gutenberg-app ul li {
    list-style-type: disc;
}
#my-first-gutenberg-app ul {
    padding-left: 20px;
}
#my-first-gutenberg-app .components-search-control__input {
    height: 36px;
    margin-left: 0;
}
#my-first-gutenberg-app .list-controls {
    display: flex;
    width: 100%;
}
#my-first-gutenberg-app .list-controls .components-search-control {
```

```
    flex-grow: 1;
    margin-right: 8px;
}
```

my-first-gutenberg-app.php:

```
<?php
/**
 * Plugin Name: My first Gutenberg App
 */
 

function my_admin_menu() {
    // Create a new admin page for our app.
    add_menu_page(
        __('My first Gutenberg app', 'gutenberg'),
        __('My first Gutenberg app', 'gutenberg'),
        'manage_options',
        'my-first-gutenberg-app',
        function () {
            echo '
                <h2>Pages</h2>
                <div id="my-first-gutenberg-app"></div>
            ';
        },
        'dashicons-schedule',
        3
    );
}

add_action( 'admin_menu', 'my_admin_menu' );

function load_custom_wp_admin_scripts( $hook ) {
    // Load only on ?page=my-first-gutenberg-app.
    if ( 'toplevel_page_my-first-gutenberg-app' !== $hook ) {
        return;
    }

    // Load the required WordPress packages.

    // Automatically load imported dependencies and assets version.
    $asset_file = include plugin_dir_path( __FILE__ ) . 'build/index.asset';

    // Enqueue CSS dependencies.
    foreach ( $asset_file['dependencies'] as $style ) {
        wp_enqueue_style( $style );
    }

    // Load our app.js.
    wp_register_script(
        'my-first-gutenberg-app',
        plugins_url( 'build/index.js', __FILE__ ),
        $asset_file['dependencies'],
        $asset_file['version']
    )
}
```

```

);
wp_enqueue_script( 'my-first-gutenberg-app' );

// Load our style.css.
wp_register_style(
    'my-first-gutenberg-app',
    plugins_url( 'style.css', __FILE__ ),
    array(),
    $asset_file['version']
);
wp_enqueue_style( 'my-first-gutenberg-app' );
}

add_action( 'admin_enqueue_scripts', 'load_custom_wp_admin_scripts' );

```

package.json:

```
{
  "name": "09-code-data-basics-esnext",
  "version": "1.1.0",
  "private": true,
  "description": "My first Gutenberg App",
  "author": "The WordPress Contributors",
  "license": "GPL-2.0-or-later",
  "keywords": [
    "WordPress",
    "block"
  ],
  "homepage": "https://github.com/WordPress/gutenberg-examples/",
  "repository": "git+https://github.com/WordPress/gutenberg-examples.git",
  "bugs": {
    "url": "https://github.com/WordPress/gutenberg-examples/issues"
  },
  "main": "build/index.js",
  "devDependencies": {
    "@wordpress/scripts": "^24.0.0"
  },
  "scripts": {
    "build": "wp-scripts build",
    "format": "wp-scripts format",
    "lint:js": "wp-scripts lint-js",
    "packages-update": "wp-scripts packages-update",
    "start": "wp-scripts start"
  }
}
```

Setting up the build pipeline

This tutorial will proceed assuming the reader is familiar with ESNext syntax and the concept of build tools (like webpack). If that sounds confusing, you may want to review the [Getting started with JavaScript Build Setup](#) first.

To install the build tool, navigate to the plugin directory using your terminal and run `npm install`.

Once all the dependencies are in place, all that's left is to run `npm start` and voila! A watcher will run in the terminal. You can then edit away in your text editor; after each save, it will automatically build.

Testing if it worked

If you now go to the Plugins page, you should see a plugin called **My first Gutenberg App**. Go ahead and activate it. A new menu item labeled *My first Gutenberg app* should show up. Once you click it, you will see a page that says *Hello from JavaScript!*:

Pages

Hello from JavaScript!

Congratulations! You are now ready to start building the app!

What's next?

- Previous part: [Introduction](#)
- Next part: [Building a basic list of pages](#)
- (optional) Review the [finished app](#) in the block-development-examples repository

First published

February 9, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Setup”](#)

[Previous](#) [Create your First App with Gutenberg Data](#) [Data](#) [Previous: Create your First App with Gutenberg Data](#)

[Next](#) [Building a list of pages](#) [Next: Building a list of pages](#)

Building a list of pages

In this article

Table of Contents

- [Step 1: Build the PagesList component](#)
- [Step 2: Fetch the data](#)
- [Step 3: Turn it into a table](#)

- [Step 4: Add a search box](#)
 - [Using core-data instead vs calling the API directly](#)
- [Step 5: Loading Indicator](#)
 - [Wiring it all together](#)
- [What's next?](#)

[↑ Back to top](#)

In this part, we will build a filterable list of all WordPress pages. This is what the app will look like at the end of this section:

Pages

Let's see how we can get there step by step.

Step 1: Build the PagesList component

Let's start by building a minimal React component to display the list of pages:

```
function MyFirstApp() {
  const pages = [{ id: 'mock', title: 'Sample page' }]
  return <PagesList pages={ pages } />;
}

function PagesList( { pages } ) {
  return (
    <ul>
      { pages?.map( page => (
        <li key={ page.id }>
          { page.title }
        </li>
      ) ) }
    </ul>
  )
}
```

```
) ;  
}
```

Note that this component does not fetch any data yet, only presents the hardcoded list of pages. When you refresh the page, you should see the following:

Pages

- Sample page

Step 2: Fetch the data

The hard-coded sample page isn't very useful. We want to display your actual WordPress pages so let's fetch the actual list of pages from the [WordPress REST API](#).

Before we start, let's confirm we actually have some pages to fetch. Within WPAdmin, Navigate to Pages using the sidebar menu and ensure it shows at least four or five Pages:

The screenshot shows the WordPress admin sidebar on the left with various menu items like Dashboard, My first Gutenberg app, Posts, Media, Pages (which is selected and highlighted in blue), All Pages, Add New, Comments, Appearance, Plugins (with a red notification badge '3'), Users, Tools, Settings, Gutenberg, and Collapse menu. The main content area is titled 'Pages' with a 'Add New' button. It shows a list of published pages: 'All (5) | Published (5) | Trash (1)'. There are buttons for 'Bulk actions', 'Apply', 'All dates', and 'Filter'. The list includes: 'Title', 'About us', 'Contact', 'Home', 'Privacy Policy — Privacy Policy Page', 'Team', and another 'Title' entry. At the bottom of the list area are 'Bulk actions' and 'Apply' buttons.

If it doesn't, go ahead and create a few pages – you can use the same titles as on the screenshot above. Be sure to *publish* and not just *save* them.

Now that we have the data to work with, let's dive into the code. We will take advantage of the [@wordpress/core-data](#) package which provides resolvers, selectors, and actions to work with the WordPress core API. `@wordpress/core-data` builds on top of the [@wordpress/data](#) package.

To fetch the list of pages, we will use the [getEntityRecords](#) selector. In broad strokes, it will issue the correct API request, cache the results, and return the list of the records we need. Here's how to use it:

```
wp.data.select('core').getEntityRecords('postType', 'page')
```

If you run that following snippet in your browser's dev tools, you will see it returns `null`. Why? The pages are only requested by the `getEntityRecords` resolver after first running the `selector`. If you wait a moment and re-run it, it will return the list of all pages.

Note: To run this type of command directly make sure your browser is displaying an instance of the block editor (any page will do). Otherwise the `select('core')` function won't be available, and you'll get an error.

Similarly, the `MyFirstApp` component needs to re-run the selector once the data is available. That's exactly what the `useSelect` hook does:

```
import { useSelect } from '@wordpress/data';
import { store as coreDataStore } from '@wordpress/core-data';

function MyFirstApp() {
    const pages = useSelect(
        select =>
            select( coreDataStore ).getEntityRecords( 'postType', 'page' )
        []
    );
    // ...
}

function PagesList({ pages }) {
    // ...
    <li key={page.id}>
        {page.title.rendered}
    </li>
    // ...
}
```

Note that we use an `import` statement inside `index.js`. This enables the plugin to automatically load the dependencies using `wp_enqueue_script`. Any references to `coreDataStore` are compiled to the same `wp.data` reference we use in browser's devtools.

`useSelect` takes two arguments: a callback and dependencies. In broad strokes, it re-runs the callback whenever either the dependencies or the underlying data store changes. You can learn more about [useSelect](#) in the [data module documentation](#).

Putting it together, we get the following code:

```
import { useSelect } from '@wordpress/data';
import { store as coreDataStore } from '@wordpress/core-data';
import { decodeEntities } from '@wordpress/html-entities';

function MyFirstApp() {
    const pages = useSelect(
        select =>
            select( coreDataStore ).getEntityRecords( 'postType', 'page' )
        []
    );
    return <PagesList pages={ pages }/>;
}

function PagesList( { pages } ) {
    return (
        <ul>
            { pages?.map( page => (
                <li key={ page.id }>
                    { decodeEntities( page.title.rendered ) }
                </li>
            ) ) }
    )
}
```

```
        </ul>
    )
}
```

Note that post title may contain HTML entities like á, so we need to use the [decodeEntities](#) function to replace them with the symbols they represent like á.

Refreshing the page should display a list similar to this one:

Pages

- Team
- Contact
- Home
- About us
- Privacy Policy

Step 3: Turn it into a table

```
function PagesList( { pages } ) {
  return (
    <table className="wp-list-table widefat fixed striped table-view-l
      <thead>
        <tr>
          <th>Title</th>
        </tr>
      </thead>
      <tbody>
        { pages?.map( page => (
          <tr key={ page.id }>
            <td>{ decodeEntities( page.title.rendered ) }</td>
          </tr>
        ) ) }
      </tbody>
    </table>
  );
}
```

Pages

Title
Team
Contact
Home
About us
Privacy Policy

Step 4: Add a search box

The list of pages is short for now; however, the longer it grows, the harder it is to work with. WordPress admins typically solves this problem with a search box – let's implement one too!

Let's start by adding a search field:

```
import { useState } from 'react';
import { SearchControl } from '@wordpress/components';

function MyFirstApp() {
    const [searchTerm, setSearchTerm] = useState( '' );
    // ...
    return (
        <div>
            <SearchControl
                onChange={ setSearchTerm }
                value={ searchTerm }
            />
            {/* ... */}
        </div>
    )
}
```

Note that instead of using an `input` tag, we took advantage of the [SearchControl](#) component. This is what it looks like:

Pages



The screenshot shows a search interface with a search bar at the top containing the placeholder "Search". Below the search bar is a list of page titles. The list includes "Title", "Team", "Contact", "Home", "About us", and "Privacy Policy". Each item in the list is contained within a separate row.

The field starts empty, and the contents are stored in the `searchTerm` state value. If you aren't familiar with the [useState](#) hook, you can learn more in [React's documentation](#).

We can now request only the pages matching the `searchTerm`.

After checking with the [WordPress API documentation](#), we see that the [/wp/v2/pages](#) endpoint accepts a `search` query parameter and uses it to *limit results to those matching a string*. But how can we use it? We can pass custom query parameters as the third argument to `getEntityRecords` as below:

```
wp.data.select( 'core' ).getEntityRecords( 'postType', 'page', { search: '' } )
```

Running that snippet in your browser's dev tools will trigger a request to `/wp/v2/pages?search=home` instead of just `/wp/v2/pages`.

Let's mirror this in our `useSelect` call as follows:

```
import { useSelect } from '@wordpress/data';
import { store as coreDataStore } from '@wordpress/core-data';

function MyFirstApp() {
    // ...
    const { pages } = useSelect( select => {
        const query = {};
        if ( searchTerm ) {
            query.search = searchTerm;
        }
        return {
            pages: select( coreDataStore ).getEntityRecords( 'postType', 'page', { search: searchTerm } ),
        }, [searchTerm] );
    }
    // ...
}
```

The `searchTerm` is now used as a `search` query parameter when provided. Note that `searchTerm` is also specified inside the list of `useSelect` dependencies to make sure `getEntityRecords` is re-run when the `searchTerm` changes.

Finally, here's how `MyFirstApp` looks once we wire it all together:

```
import { useState } from 'react';
import { createRoot } from 'react-dom';
import { SearchControl } from '@wordpress/components';
import { useSelect } from '@wordpress/data';
import { store as coreDataStore } from '@wordpress/core-data';

function MyFirstApp() {
    const [searchTerm, setSearchTerm] = useState( '' );
    const pages = useSelect( select => {
        const query = {};
        if ( searchTerm ) {
            query.search = searchTerm;
        }
        return select( coreDataStore ).getEntityRecords( 'postType', 'page'
    }, [searchTerm] );

    return (
        <div>
            <SearchControl
                onChange={ setSearchTerm }
                value={ searchTerm }
            />
            <PagesList pages={ pages }/>
        </div>
    )
}
```

Voila! We can now filter the results:



Using core-data instead vs calling the API directly

Let's take a pause for a moment to consider the downsides of an alternative approach we could have taken – working with the API directly. Imagine we sent the API requests directly:

```
import apiFetch from '@wordpress/api-fetch';
function MyFirstApp() {
```

```

// ...
const [pages, setPages] = useState( [] );
useEffect( () => {
    const url = '/wp-json/wp/v2/pages?search=' + searchTerm;
    apiFetch( { url } )
        .then( setPages )
}, [searchTerm] );
// ...
}

```

Working outside of core-data, we would need to solve two problems here.

Firstly, out-of-order updates. Searching for „About” would trigger five API requests filtering for A, Ab, Abo, Abou, and About. These requests could finish in a different order than they started. It is possible that `search=A` would resolve after `_search=About_` and thus we'd display the wrong data.

Gutenberg data helps by handling the asynchronous part behind the scenes. `useSelect` remembers the most recent call and returns only the data we expect.

Secondly, every keystroke would trigger an API request. If you typed `About`, deleted it, and retyped it, it would issue 10 requests in total even though we could reuse the data.

Gutenberg data helps by caching the responses to API requests triggered by `getEntityRecords()` and reuses them on subsequent calls. This is especially important when other components rely on the same entity records.

All in all, the utilities built into core-data are designed to solve the typical problems so that you can focus on your application instead.

Step 5: Loading Indicator

There is one problem with our search feature. We can't be quite sure whether it's still searching or showing no results:



A few messages like `Loading...` or `No results` would clear it up. Let's implement them! First, `PagesList` has to be aware of the current status:

```

import { SearchControl, Spinner } from '@wordpress/components';
function PagesList( { hasResolved, pages } ) {
    if ( !hasResolved ) {
        return <Spinner/>
    }
}

```

```

        if ( !pages?.length ) {
            return <div>No results</div>
        }
        // ...
    }

function MyFirstApp() {
    // ...

    return (
        <div>
            // ...
            <PagesList hasResolved={ hasResolved } pages={ pages }/>
        </div>
    )
}

```

Note that instead of building a custom loading indicator, we took advantage of the [Spinner](#) component.

We still need to know whether the pages selector `hasResolved` or not. We can find out using the `hasFinishedResolution` selector:

```
wp.data.select('core').hasFinishedResolution( 'getEntityRecords' ,
[ 'postType', 'page', { search: 'home' } ] )
```

It takes the name of the selector and the *exact same arguments you passed to that selector* and returns either `true` if the data was already loaded or `false` if we're still waiting. Let's add it to `useSelect`:

```

import { useSelect } from '@wordpress/data';
import { store as coreDataStore } from '@wordpress/core-data';

function MyFirstApp() {
    // ...
    const { pages, hasResolved } = useSelect( select => {
        // ...
        return {
            pages: select( coreDataStore ).getEntityRecords( 'postType' ,
                hasResolved:
                    select( coreDataStore ).hasFinishedResolution( 'getEntityR
            }
        }, [searchTerm] );
    }
    // ...
}

```

There is just one last problem. It is easy to make a typo and end up passing different arguments to `getEntityRecords` and `hasFinishedResolution`. It is critical that they are identical. We can remove this risk by storing the arguments in a variable:

```

import { useSelect } from '@wordpress/data';
import { store as coreDataStore } from '@wordpress/core-data';
function MyFirstApp() {

```

```

// ...
const { pages, hasResolved } = useSelect( select => {
    // ...
    const selectorArgs = [ 'postType', 'page', query ];
    return {
        pages: select( coreDataStore ).getEntityRecords( ...selectorArgs ),
        hasResolved: select( coreDataStore ).hasFinishedResolution( 'getEntityRecords' ),
    }
}, [searchTerm] );
// ...
}

```

And voilà! That's it.

Wiring it all together

All the pieces are in place, great! Here's the complete JavaScript code of our app:

```

import { useState } from 'react';
import { createRoot } from 'react-dom';
import { SearchControl, Spinner } from '@wordpress/components';
import { useSelect } from '@wordpress/data';
import { store as coreDataStore } from '@wordpress/core-data';
import { decodeEntities } from '@wordpress/html-entities';

function MyFirstApp() {
    const [ searchTerm, setSearchTerm ] = useState( '' );
    const { pages, hasResolved } = useSelect(
        ( select ) => {
            const query = {};
            if ( searchTerm ) {
                query.search = searchTerm;
            }
            const selectorArgs = [ 'postType', 'page', query ];
            return {
                pages: select( coreDataStore ).getEntityRecords(
                    ...selectorArgs
                ),
                hasResolved: select( coreDataStore ).hasFinishedResolution(
                    'getEntityRecords',
                    selectorArgs
                ),
            };
        },
        [ searchTerm ]
    );

    return (
        <div>
            <SearchControl onChange={ setSearchTerm } value={ searchTerm } />
            <PagesList hasResolved={ hasResolved } pages={ pages } />
        </div>
    );
}

```

```

    );
}

function PagesList( { hasResolved, pages } ) {
  if ( ! hasResolved ) {
    return <Spinner />;
  }
  if ( ! pages?.length ) {
    return <div>No results</div>;
  }

  return (
    <table className="wp-list-table widefat fixed striped table-view-l
      <thead>
        <tr>
          <td>Title</td>
        </tr>
      </thead>
      <tbody>
        { pages?.map( ( page ) => (
          <tr key={ page.id }>
            <td>{ decodeEntities( page.title.rendered ) }</td>
          </tr>
        ) ) }
      </tbody>
    </table>
  );
}

const root = createRoot(
  document.querySelector( '#my-first-gutenberg-app' )
);
window.addEventListener(
  'load',
  function () {
    root.render(
      <MyFirstApp />
    );
  },
  false
);

```

All that's left is to refresh the page and enjoy the brand new status indicator:

Pages

[nosuchpage](#)

X

Pages

[nosuchpage](#)

X

No results

What's next?

- Previous part: [Setup](#)
- Next part: [Building an edit form](#)
- (optional) Review the [finished app](#) in the block-development-examples repository

First published

February 9, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Building a list of pages”](#)

[Previous Setup](#) [Previous: Setup](#)

[Next Building an edit form](#) [Next: Building an edit form](#)

Building an edit form

In this article

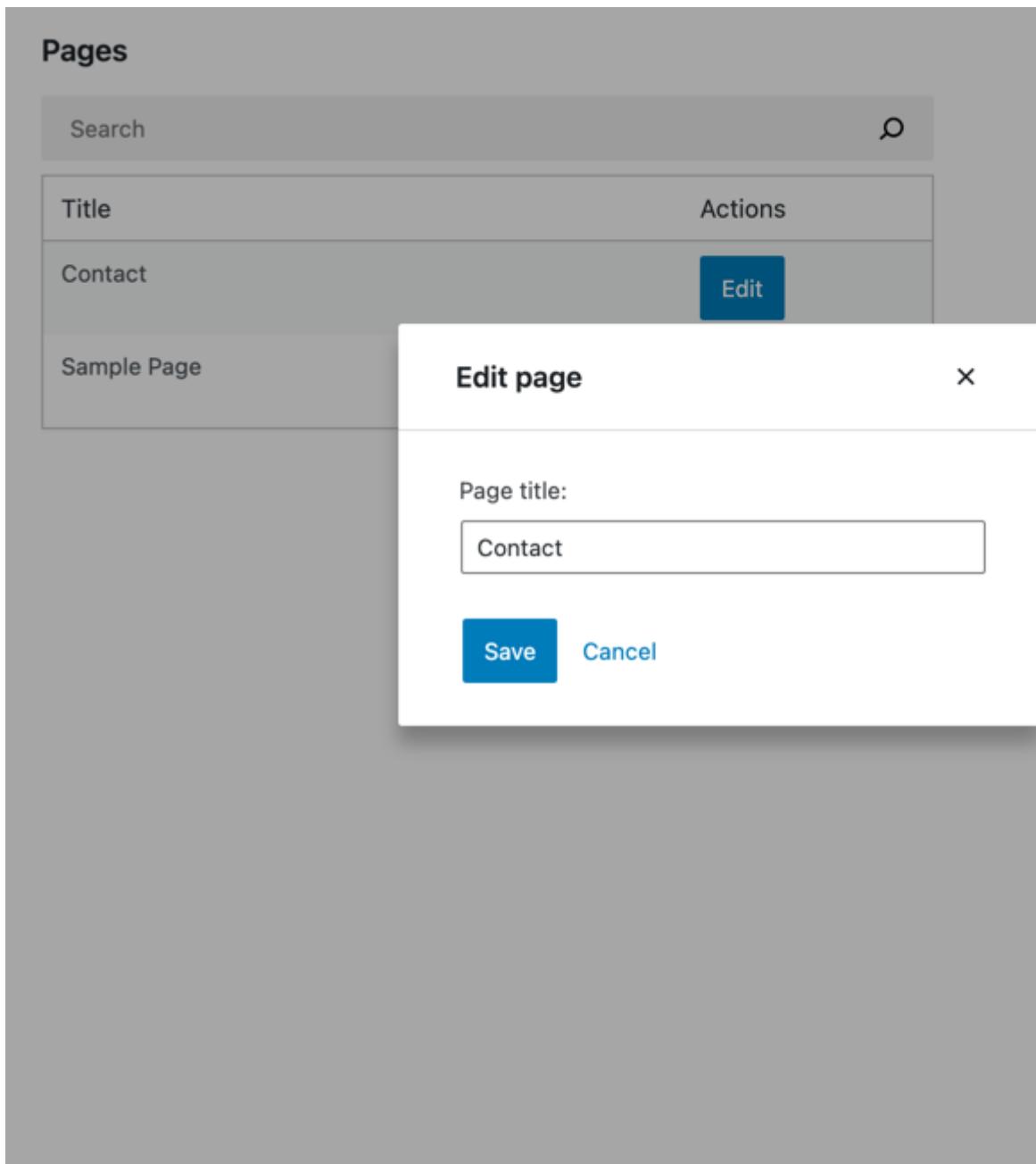
[Table of Contents](#)

- [Step 1: Add an Edit button](#)
- [Step 2: Display an Edit form](#)
- [Step 3: Populate the form with page details](#)
- [Step 4: Making the Page title field editable](#)
- [Step 5: Saving the form data](#)
- [Step 6: Handle errors](#)

- [Step 7: Status indicator](#)
- [Wiring it all together](#)
- [What's next?](#)

[↑ Back to top](#)

This part is about adding an *Edit* feature to our app. Here's a glimpse of what we're going to build:



[**Step 1: Add an Edit button**](#)

We can't have an *Edit* form without an *Edit* button, so let's start by adding one to our `PagesList` component:

```

import { Button } from '@wordpress/components';
import { decodeEntities } from '@wordpress/html-entities';

const PageEditButton = () => (
    <Button variant="primary">
        Edit
    </Button>
)

function PagesList( { hasResolved, pages } ) {
    if ( ! hasResolved ) {
        return <Spinner />;
    }
    if ( ! pages?.length ) {
        return <div>No results</div>;
    }

    return (
        <table className="wp-list-table widefat fixed striped table-view-l
            <thead>
                <tr>
                    <td>Title</td>
                    <td style={{width: 120}}>Actions</td>
                </tr>
            </thead>
            <tbody>
                { pages?.map( ( page ) => (
                    <tr key={page.id}>
                        <td>{ decodeEntities( page.title.rendered ) }</td>
                        <td>
                            <PageEditButton pageId={ page.id } />
                        </td>
                    </tr>
                ) ) }
            </tbody>
        </table>
    );
}

```

The only change in `PagesList` is the additional column labeled *Actions*:

Pages

Title	Actions
Contact	<button>Edit</button>
Sample Page	<button>Edit</button>

Step 2: Display an Edit form

Our button looks nice but doesn't do anything yet. To display an edit form, we need to have one first – let's create it:

```
import { Button, TextControl } from '@wordpress/components';
function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
    return (
        <div className="my-gutenberg-form">
            <TextControl
                value=''
                label='Page title:'
            />
            <div className="form-buttons">
                <Button onClick={ onSaveFinished } variant="primary">
                    Save
                </Button>
                <Button onClick={ onCancel } variant="tertiary">
                    Cancel
                </Button>
            </div>
        </div>
    );
}
```

Now let's make the button display the form we just created. As this tutorial is not focused on web design, we will wire the two together using a component that requires the least amount of code: [Modal](#). Let's update `PageEditButton` accordingly:

```
import { Button, Modal, TextControl } from '@wordpress/components';

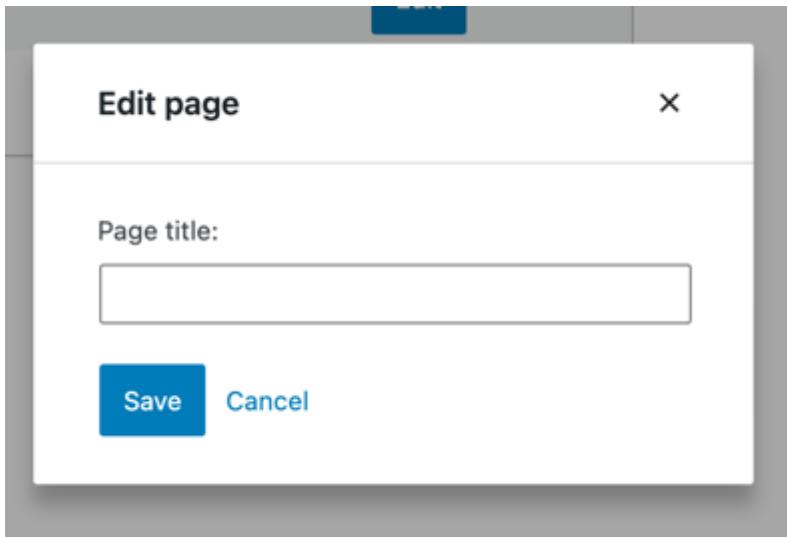
function PageEditButton({ pageId }) {
    const [ isOpen, setOpen ] = useState( false );
    const openModal = () => setOpen( true );
    const closeModal = () => setOpen( false );
    return (
        <>
            <Button
```

```

        onClick={ openModal }
        variant="primary"
      >
    Edit
  </Button>
  { isOpen && (
    <Modal onRequestClose={ closeModal } title="Edit page">
      <EditPageForm
        pageId={pageId}
        onCancel={closeModal}
        onSaveFinished={closeModal}
      />
    </Modal>
  ) }
</>
)
}

```

When you click the *Edit* button now, you should see the following modal:



Great! We now have a basic user interface to work with.

Step 3: Populate the form with page details

We want the `EditPageForm` to display the title of the currently edited page. You may have noticed that it doesn't receive a `page` prop, only `pageId`. That's okay. Gutenberg Data allows us to easily access entity records from any component.

In this case, we need to use the [getEntityRecord](#) selector. The list of records is already available thanks to the `getEntityRecords` call in `MyFirstApp`, so there won't even be any additional HTTP requests involved – we'll get the cached record right away.

Here's how you can try it in your browser's dev tools:

```
wp.data.select( 'core' ).getEntityRecord( 'postType', 'page', 9 ); // Rep
```

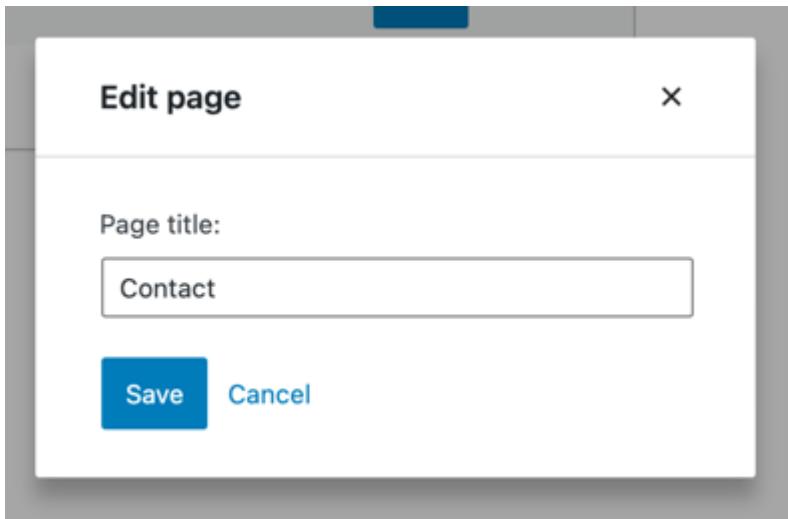
Let's update `EditPageForm` accordingly:

```

function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
  const page = useSelect(
    select => select( coreDataStore ).getEntityRecord( 'postType', 'pa
      [pageId]
  );
  return (
    <div className="my-gutenberg-form">
      <TextControl
        label='Page title:'
        value={ page.title.rendered }
      />
      { /* ... */ }
    </div>
  );
}

```

Now it should look like this:



Step 4: Making the Page title field editable

There's one problem with our *Page title* field: you can't edit it. It receives a fixed `value` but doesn't update it when typing. We need an `onChange` handler.

You may have seen a pattern similar to this one in other React apps. It's known as a [“controlled component”](#):

```

function VanillaReactForm({ initialTitle }) {
  const [title, setTitle] = useState( initialTitle );
  return (
    <TextControl
      value={ title }
      onChange={ setTitle }
    />
  );
}

```

Updating entity records in Gutenberg Data is similar but instead of using `setTitle` to store in local (component level) state, we use the `editEntityRecord` action which stores the updates in the *Redux* state. Here's how you can try it out in your browser's dev tools:

```
// We need a valid page ID to call editEntityRecord, so let's get the first
const pageId = wp.data.select( 'core' ).getEntityRecords( 'postType', 'page' )

// Update the title
wp.data.dispatch( 'core' ).editEntityRecord( 'postType', 'page', pageId, {
```

At this point, you may ask *how* is `editEntityRecord` better than `useState`? The answer is that it offers a few features you wouldn't otherwise get.

Firstly, we can save the changes as easily as we retrieve the data and ensure that all caches will be correctly updated.

Secondly, the changes applied via `editEntityRecord` are easily undo-able via the `undo` and `redo` actions.

Lastly, because the changes live in the *Redux* state, they are “global” and can be accessed by other components. For example, we could make the `PagesList` display the currently edited title.

To that last point, let's see what happens when we use `getEntityRecord` to access the entity record we just updated:

```
wp.data.select( 'core' ).getEntityRecord( 'postType', 'page', pageId ).title
```

It doesn't reflect the edits. What's going on?

Well, `<PagesList />` renders the data returned by `getEntityRecord()`. If `getEntityRecord()` reflected the updated title, then anything the user types in the `TextControl` would be immediately displayed inside `<PagesList />`, too. This is not what we want. The edits shouldn't leak outside the form until the user decides to save them.

Gutenberg Data solves this problem by making a distinction between *Entity Records* and *Edited Entity Records*. *Entity Records* reflect the data from the API and ignore any local edits, while *Edited Entity Records* also have all the local edits applied on top. Both co-exist in the *Redux* state at the same time.

Let's see what happens if we call `getEditedEntityRecord`:

```
wp.data.select( 'core' ).getEditedEntityRecord( 'postType', 'page', pageId )
// "updated title"
```

```
wp.data.select( 'core' ).getEntityRecord( 'postType', 'page', pageId ).title
// { "rendered": "<original, unchanged title>", "raw": "..." }
```

As you can see, the `title` of an Entity Record is an object, but the `title` of an Edited Entity record is a string.

This is no accident. Fields like `title`, `excerpt`, and `content` may contain [shortcodes](#) or [dynamic blocks](#), which means they can only be rendered on the server. For such fields, the REST API exposes both the `raw` markup and the `rendered` string. For example, in the block editor,

`content.rendered` could be used as a visual preview, and `content.raw` could be used to populate the code editor.

So why is the `content` of an Edited Entity Record a string? Since Javascript is not able to properly render arbitrary block markup, it stores only the `raw` markup without the `rendered` part. And since that's a string, the entire field becomes a string.

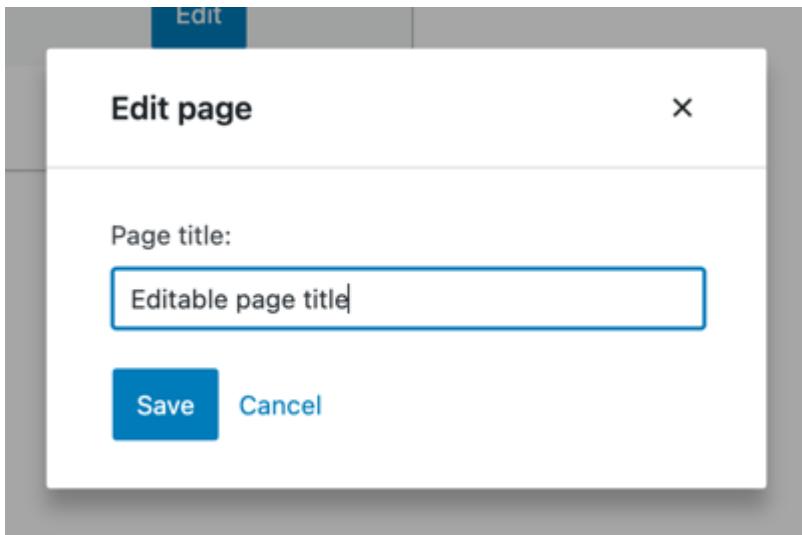
We can now update `EditPageForm` accordingly. We can access the actions using the [`useDispatch`](#) hook similarly to how we use `useSelect` to access selectors:

```
import { useDispatch } from '@wordpress/data';

function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
    const page = useSelect(
        select => select( coreDataStore ).getEditedEntityRecord( 'postType'
            [ pageId ]
    );
    const { editEntityRecord } = useDispatch( coreDataStore );
    const handleChange = ( title ) => editEntityRecord( 'postType', 'page'
        title
    );
    return (
        <div className="my-gutenberg-form">
            <TextControl
                label="Page title:"
                value={ page.title }
                onChange={ handleChange }
            />
            <div className="form-buttons">
                <Button onClick={ onSaveFinished } variant="primary">
                    Save
                </Button>
                <Button onClick={ onCancel } variant="tertiary">
                    Cancel
                </Button>
            </div>
        </div>
    );
}
```

We added an `onChange` handler to keep track of edits via the `editEntityRecord` action and then changed the selector to `getEditedEntityRecord` so that `page.title` always reflects the changes.

This is what it looks like now:



Step 5: Saving the form data

Now that we can edit the page title let's also make sure we can save it. In Gutenberg data, we save changes to the WordPress REST API using the `saveEditedEntityRecord` action. It sends the request, processes the result, and updates the cached data in the Redux state.

Here's an example you may try in your browser's dev tools:

```
// Replace 9 with an actual page ID
wp.data.dispatch( 'core' ).editEntityRecord( 'postType', 'page', 9, { title: 'updated title' } )
wp.data.dispatch( 'core' ).saveEditedEntityRecord( 'postType', 'page', 9 )
```

The above snippet saved a new title. Unlike before, `getEntityRecord` now reflects the updated title:

```
// Replace 9 with an actual page ID
wp.data.select( 'core' ).getEntityRecord( 'postType', 'page', 9 ).title
// "updated title"
```

Entity records are updated to reflect any saved changes right after the REST API request is finished.

This is how the `EditPageForm` looks like with a working `Save` button:

```
function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
    // ...
    const { saveEditedEntityRecord } = useDispatch( coreDataStore );
    const handleSave = () => saveEditedEntityRecord( 'postType', 'page', pageId );

    return (
        <div className="my-gutenberg-form">
            {/* ... */}
            <div className="form-buttons">
                <Button onClick={ handleSave } variant="primary">
                    Save
                </Button>
                {/* ... */}
            </div>
        </div>
    );
}
```

```

        </div>
    );
}

```

It works, but there's still one thing to fix: the form modal doesn't automatically close because we never call `onSaveFinished`. Lucky for us, `saveEditedEntityRecord` returns a promise that resolves once the save operation is finished. Let's take advantage of it in `EditPageForm`:

```

function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
    // ...
    const handleSave = async () => {
        await saveEditedEntityRecord( 'postType', 'page', pageId );
        onSaveFinished();
    };
    // ...
}

```

Step 6: Handle errors

We optimistically assumed that a `save` operation would always succeed. Unfortunately, it may fail in many ways:

- The website can be down
- The update may be invalid
- The page could have been deleted by someone else in the meantime

To tell the user when any of these happens, we have to make two adjustments. We don't want to close the form modal when the update fails. The promise returned by `saveEditedEntityRecord` is resolved with an updated record only if the update actually worked. When something goes wrong, it resolves with an empty value. Let's use it to keep the modal open:

```

function EditPageForm( { pageId, onSaveFinished } ) {
    // ...
    const handleSave = async () => {
        const updatedRecord = await saveEditedEntityRecord( 'postType', 'p
        if ( updatedRecord ) {
            onSaveFinished();
        }
    };
    // ...
}

```

Great! Now, let's display an error message. The failure details can be grabbed using the `getLastEntitySaveError` selector:

```
// Replace 9 with an actual page ID
wp.data.select( 'core' ).getLastEntitySaveError( 'postType', 'page', 9 )
```

Here's how we can use it in `EditPageForm`:

```

function EditPageForm( { pageId, onSaveFinished } ) {
    // ...
    const { lastError, page } = useSelect(

```

```

        select => ({
            page: select( coreDataStore ).getEditedEntityRecord( 'postType' ),
            lastError: select( coreDataStore ).getLastEntitySaveError( 'postType' )
        }),
        [ pageId ]
    )
    // ...
    return (
        <div className="my-gutenberg-form">
            {/* ... */}
            { lastError ? (
                <div className="form-error">
                    Error: { lastError.message }
                </div>
            ) : false }
            {/* ... */}
        </div>
    );
}

```

Great! `EditPageForm` is now fully aware of errors.

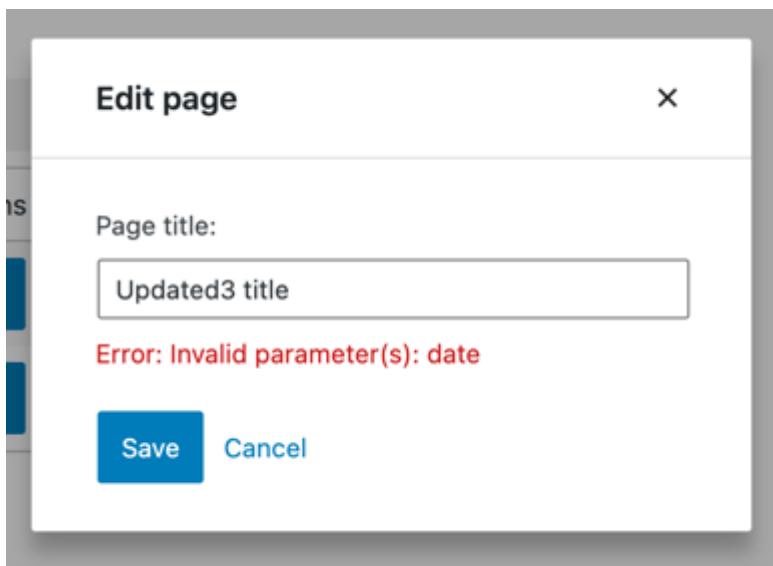
Let's see that error message in action. We'll trigger an invalid update and let it fail. The post title is hard to break, so let's set a `date` property to `-1` instead – that's a guaranteed validation error:

```

function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
    // ...
    const handleChange = ( title ) => editEntityRecord( 'postType', 'pageTitle', title )
    // ...
}

```

Once you refresh the page, open the form, change the title, and hit save, you should see the following error message:



Fantastic! We can now **restore the previous version of `handleChange`** and move on to the next step.

Step 7: Status indicator

There is one last problem with our form: no visual feedback. We can't be quite sure whether the *Save* button worked until either the form disappears or an error message shows.

We're going to clear it up and communicate two states to the user: *Saving* and *No changes detected*. The relevant selectors are `isSavingEntityRecord` and `hasEditsForEntityRecord`. Unlike `getEntityRecord`, they never issue any HTTP requests but only return the current entity record state.

Let's use them in `EditPageForm`:

```
function EditPageForm( { pageId, onSaveFinished } ) {
    // ...
    const { isSaving, hasEdits, /* ... */ } = useSelect(
        select => ({
            isSaving: select( coreDataStore ).isSavingEntityRecord( 'post' ),
            hasEdits: select( coreDataStore ).hasEditsForEntityRecord( 'post' ),
            // ...
        }),
        [ pageId ]
    )
}
```

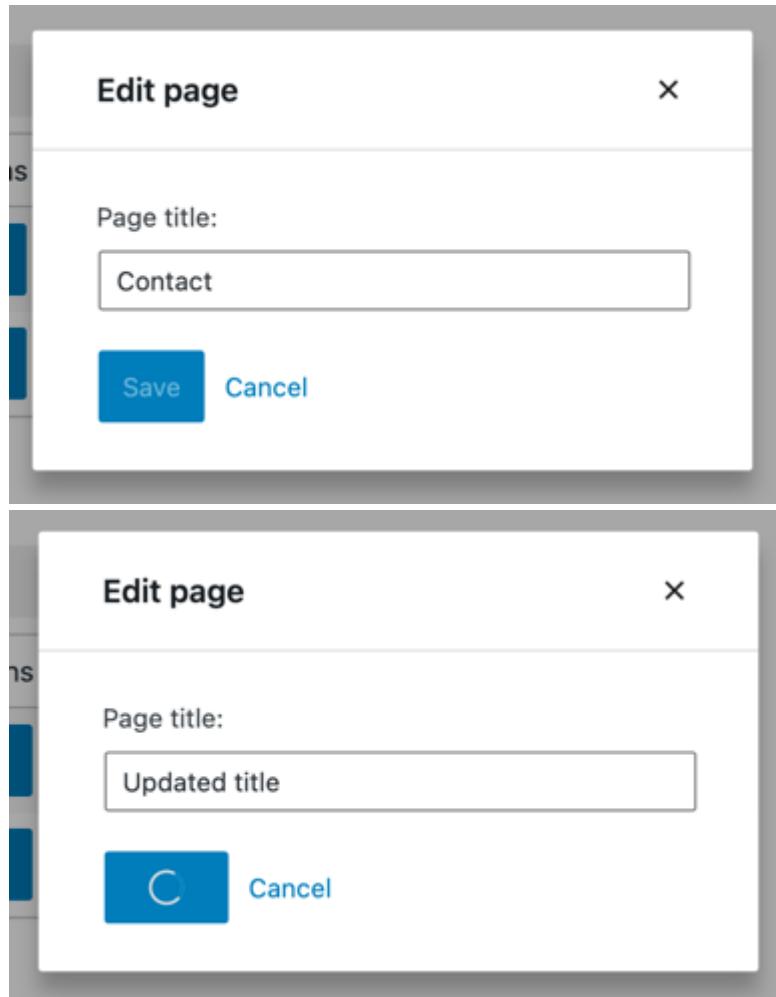
We can now use `isSaving` and `hasEdits` to display a spinner when saving is in progress and grey out the save button when there are no edits:

```
function EditPageForm( { pageId, onSaveFinished } ) {
    // ...
    return (
        // ...
        <div className="form-buttons">
            <Button onClick={ handleSave } variant="primary" disabled={ ! isSaving }>
                { isSaving ? (
                    <>
                        <Spinner/>
                        Saving
                    </>
                ) : 'Save' }
            </Button>
            <Button
                onClick={ onCancel }
                variant="tertiary"
                disabled={ isSaving }
            >
                Cancel
            </Button>
        </div>
        // ...
    );
}
```

Note that we disable the *save* button when there are no edits and when the page is currently being saved. This is to prevent the user from accidentally pressing the button twice.

Also, interrupting a *save* in progress is not supported by `@wordpress/data` so we also conditionally disabled the *cancel* button.

Here's what it looks like in action:



Wiring it all together

All the pieces are in place, great! Here's everything we built in this chapter in one place:

```
import { useDispatch } from '@wordpress/data';
import { Button, Modal, TextControl } from '@wordpress/components';

function PageEditButton( { pageId } ) {
    const [ isOpen, setOpen ] = useState( false );
    const openModal = () => setOpen( true );
    const closeModal = () => setOpen( false );
    return (
        <>
            <Button onClick={ openModal } variant="primary">
                Edit
            </Button>
            { isOpen && (
                <Modal onRequestClose={ closeModal } title="Edit page">
                    <EditPageForm
                        pageId={ pageId }>

```

```

        onCancel={ closeModal }
        onSaveFinished={ closeModal }
      />
    </Modal>
  ) )
</>
);
}

function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
  const { page, lastError, isSaving, hasEdits } = useSelect(
    ( select ) => ( {
      page: select( coreDataStore ).getEditedEntityRecord( 'postType' ),
      lastError: select( coreDataStore ).getLastEntitySaveError( 'po',
      isSaving: select( coreDataStore ).isSavingEntityRecord( 'postT
      hasEdits: select( coreDataStore ).hasEditsForEntityRecord( 'po
    } ),
    [ pageId ]
  );
}

const { saveEditedEntityRecord, editEntityRecord } = useDispatch( core
const handleSave = async () => {
  const savedRecord = await saveEditedEntityRecord( 'postType', 'page
  if ( savedRecord ) {
    onSaveFinished();
  }
};
const handleChange = ( title ) => editEntityRecord( 'postType', 'page
return (
  <div className="my-gutenberg-form">
    <TextControl
      label="Page title:"
      value={ page.title }
      onChange={ handleChange }
    />
    { lastError ? (
      <div className="form-error">Error: { lastError.message }</div>
    ) : (
      false
    ) }
    <div className="form-buttons">
      <Button
        onClick={ handleSave }
        variant="primary"
        disabled={ ! hasEdits || isSaving }
      >
        { isSaving ? (
          <>
            <Spinner/>
            Saving
          </>
        ) : 'Save' }
      </Button>
    
```

```
<Button
    onClick={ onCancel }
    variant="tertiary"
    disabled={ isSaving }
>
    Cancel
</Button>
</div>
</div>
);
}
```

What's next?

- Previous part: [Building a list of pages](#)
- Next part: Building a *New Page* form (coming soon)
- (optional) Review the [finished app](#) in the block-development-examples repository

First published

March 16, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Building an edit form”](#)

[Previous Building a list of pages](#) [Previous: Building a list of pages](#)
[Next Part 4: Building a Create page form](#) [Next: Part 4: Building a Create page form](#)

Part 4: Building a Create page form

In this article

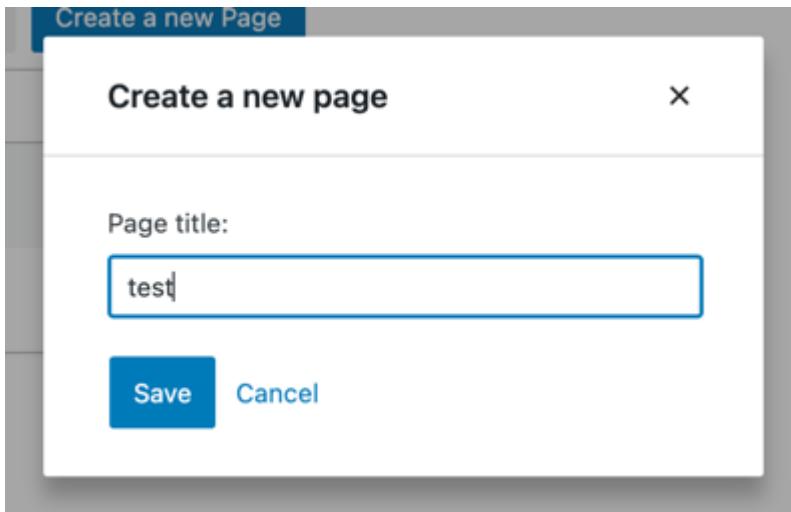
Table of Contents

- [Step 1: Add a Create a new page button](#)
- [Step 2: Extract a controlled PageForm](#)
- [Step 3: Build a CreatePageForm](#)
- [Wiring it all together](#)

- [What's next?](#)

[↑ Back to top](#)

In the [previous part](#) we created an *Edit page* feature, and in this part we will add a *Create page* feature. Here's a glimpse of what we're going to build:



Step 1: Add a Create a new page button

Let's start by building a button to display the *create page* form. It's similar to an *Edit* button we have built in the [part 3](#):

```
import { useDispatch } from '@wordpress/data';
import { Button, Modal, TextControl } from '@wordpress/components';

function CreatePageButton() {
    const [isOpen, setOpen] = useState( false );
    const openModal = () => setOpen( true );
    const closeModal = () => setOpen( false );
    return (
        <>
            <Button onClick={ openModal } variant="primary">
                Create a new Page
            </Button>
            { isOpen && (
                <Modal onRequestClose={ closeModal } title="Create a new p
                    <CreatePageForm
                        onCancel={ closeModal }
                        onSaveFinished={ closeModal }
                    />
                </Modal>
            ) }
        </>
    );
}

function CreatePageForm() {
    // Empty for now
    return <div/>;
}
```

Great! Now let's make MyFirstApp display our shiny new button:

```

function MyFirstApp() {
    // ...
    return (
        <div>
            <div className="list-controls">
                <SearchControl onChange={ setSearchTerm } value={ searchTerm } />
                <CreatePageButton/>
            </div>
            <PagesList hasResolved={ hasResolved } pages={ pages } />
        </div>
    );
}

```

The final result should look as follows:

Title	Actions
Contact	<button>Edit</button>
Sample Page	<button>Edit</button>

Step 2: Extract a controlled PageForm

Now that the button is in place, we can focus entirely on building the form. This tutorial is about managing data, so we will not build a complete page editor. Instead, the form will only contain one field: post title.

Luckily, the `EditPageForm` we built in [part three](#) already takes us 80% of the way there. The bulk of the user interface is already available, and we will reuse it in the `CreatePageForm`. Let's start by extracting the form UI into a separate component:

```

function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
    // ...
    return (
        <PageForm
            title={ page.title }
            onChangeTitle={ handleChange }
            hasEdits={ hasEdits }
            lastError={ lastError }
            isSaving={ isSaving }
            onCancel={ onCancel }
            onSave={ handleSave }
        />
    );
}

```

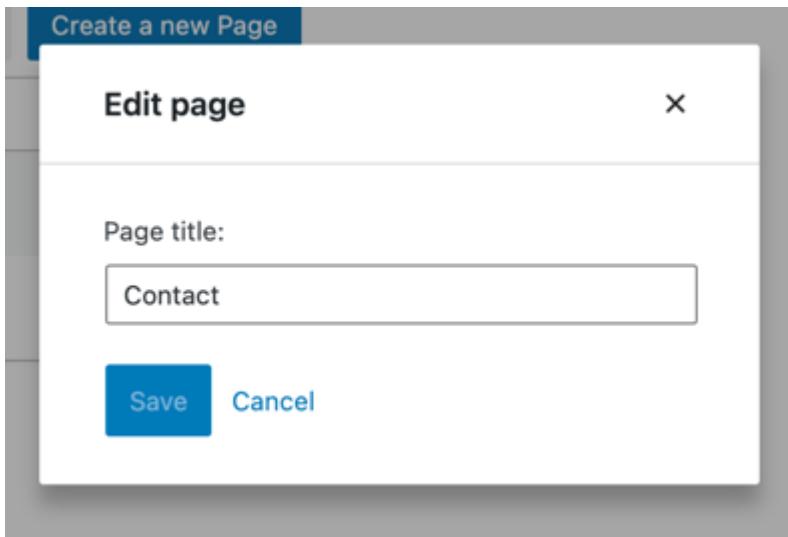
```

}

function PageForm( { title, onChangeTitle, hasEdits, lastError, isSaving,
  return (
    <div className="my-gutenberg-form">
      <TextControl
        label="Page title:"
        value={ title }
        onChange={ onChangeTitle }
      />
      { lastError ? (
        <div className="form-error">Error: { lastError.message }</div>
      ) : (
        false
      ) }
      <div className="form-buttons">
        <Button
          onClick={ onSave }
          variant="primary"
          disabled={ !hasEdits || isSaving }
        >
          { isSaving ? (
            <>
              <Spinner/>
              Saving
            </>
          ) : 'Save' }
        </Button>
        <Button
          onClick={ onCancel }
          variant="tertiary"
          disabled={ isSaving }
        >
          Cancel
        </Button>
      </div>
    </div>
  );
}

```

This code quality change should not alter anything about how the application works. Let's try to edit a page just to be sure:



Great! The edit form is still there, and now we have a building block to power the new `CreatePageForm`.

Step 3: Build a CreatePageForm

The only thing that `CreatePageForm` component must do is to provide the following seven properties needed to render the `PageForm` component:

- `title`
- `onChangeTitle`
- `hasEdits`
- `lastError`
- `isSaving`
- `onCancel`
- `onSave`

Let's see how we can do that:

Title, onChangeTitle, hasEdits

The `EditPageForm` updated and saved an existing entity record that lived in the Redux state. Because of that, we relied on the `editedEntityRecords` selector.

In case of the `CreatePageForm` however, there is no pre-existing entity record. There is only an empty form. Anything that the user types is local to that form, which means we can keep track of it using the React's `useState` hook:

```
function CreatePageForm( { onCancel, onSaveFinished } ) {
  const [title, setTitle] = useState();
  const handleChange = ( title ) => setTitle( title );
  return (
    <PageForm
      title={ title }
      onChangeTitle={ setTitle }
      hasEdits={ !!title }
      { /* ... */ }
    />
  );
}
```

```
    );
}
```

onSave, onCancel

In the `EditPageForm`, we dispatched the `saveEditedEntityRecord('postType', 'page', pageId)` action to save the edits that lived in the Redux state.

In the `CreatePageForm` however, we do not have any edits in the Redux state, nor we do have a `pageId`. The action we need to dispatch in this case is called `saveEntityRecord` (without the word *Edited* in the name) and it accepts an object representing the new entity record instead of a `pageId`.

The data passed to `saveEntityRecord` is sent via a POST request to the appropriate REST API endpoint. For example, dispatching the following action:

```
saveEntityRecord( 'postType', 'page', { title: "Test" } );
```

Triggers a POST request to the [`/wp/v2/pages` WordPress REST API](#) endpoint with a single field in the request body: `title=Test`.

Now that we know more about `saveEntityRecord`, let's use it in `CreatePageForm`.

```
function CreatePageForm( { onSaveFinished, onCancel } ) {
    // ...
    const { saveEntityRecord } = useDispatch( coreDataStore );
    const handleSave = async () => {
        const savedRecord = await saveEntityRecord(
            'postType',
            'page',
            { title }
        );
        if ( savedRecord ) {
            onSaveFinished();
        }
    };
    return (
        <PageForm
            { /* ... */ }
            onSave={ handleSave }
            onCancel={ onCancel }
        />
    );
}
```

There is one more detail to address: our newly created pages are not yet picked up by the `PagesList`. Accordingly to the REST API documentation, the `/wp/v2/pages` endpoint creates (POST requests) pages with `status=draft` by default, but `returns` (GET requests) pages with `status=publish`. The solution is to pass the `status` parameter explicitly:

```
function CreatePageForm( { onSaveFinished, onCancel } ) {
    // ...
    const { saveEntityRecord } = useDispatch( coreDataStore );
    const handleSave = async () => {
```

```

        const savedRecord = await saveEntityRecord(
            'postType',
            'page',
            { title, status: 'publish' }
        );
        if ( savedRecord ) {
            onSaveFinished();
        }
    };
    return (
        <PageForm
            { /* ... */ }
            onSave={ handleSave }
            onCancel={ onCancel }
        />
    );
}

```

Go ahead and apply that change to your local `CreatePageForm` component, and let's tackle the remaining two props.

lastError, isSaving

The `EditPageForm` retrieved the error and progress information via the `getLastEntitySaveError` and `isSavingEntityRecord` selectors. In both cases, it passed the following three arguments: (`'postType'`, `'page'`, `pageId`).

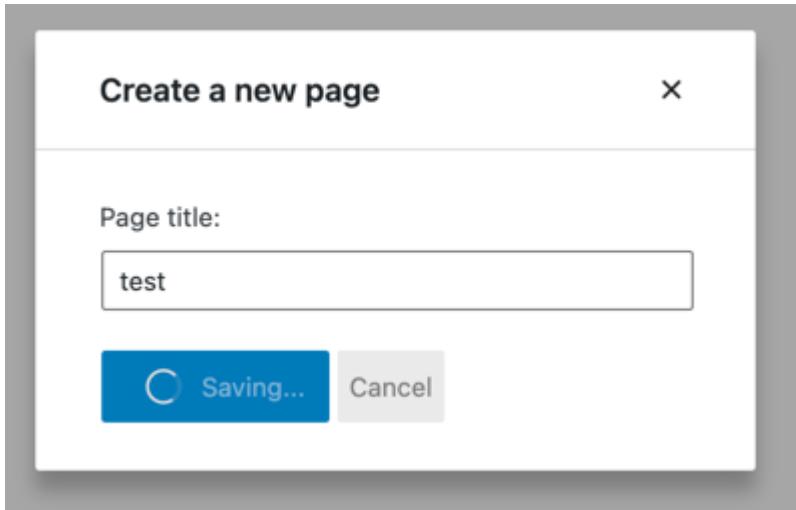
In `CreatePageForm` however, we do not have a `pageId`. What now? We can skip the `pageId` argument to retrieve the information about the entity record without any id – this will be the newly created one. The `useSelect` call is thus very similar to the one from `EditPageForm`:

```

function CreatePageForm( { onCancel, onSaveFinished } ) {
    // ...
    const { lastError, isSaving } = useSelect(
        ( select ) => ( {
            // Notice the missing pageId argument:
            lastError: select( coreDataStore )
                .getLastEntitySaveError( 'postType', 'page' ),
            // Notice the missing pageId argument
            isSaving: select( coreDataStore )
                .isSavingEntityRecord( 'postType', 'page' ),
        } ),
        []
    );
    // ...
    return (
        <PageForm
            { /* ... */ }
            lastError={ lastError }
            isSaving={ isSaving }
        />
    );
}

```

And that's it! Here's what our new form looks like in action:



Pages

Pages	
Title	Actions
test	<button>Edit</button>
Contact	<button>Edit</button>
Sample Page	<button>Edit</button>

Wiring it all together

Here's everything we built in this chapter in one place:

```
function CreatePageForm( { onCancel, onSaveFinished } ) {
  const [title, setTitle] = useState();
  const { lastError, isSaving } = useSelect(
    ( select ) => ( {
      lastError: select( coreDataStore )
        .getLastEntitySaveError( 'postType', 'page' ),
      isSaving: select( coreDataStore )
        .isSavingEntityRecord( 'postType', 'page' ),
    } ),
    []
  );
  const { saveEntityRecord } = useDispatch( coreDataStore );
  const handleSave = async () => {
```

```

        const savedRecord = await saveEntityRecord(
            'postType',
            'page',
            { title, status: 'publish' }
        );
        if ( savedRecord ) {
            onSaveFinished();
        }
    };

    return (
        <PageForm
            title={ title }
            onChangeTitle={ setTitle }
            hasEdits={ !!title }
            onSave={ handleSave }
            lastError={ lastError }
            onCancel={ onCancel }
            isSaving={ isSaving }
        />
    );
}

function EditPageForm( { pageId, onCancel, onSaveFinished } ) {
    const { page, lastError, isSaving, hasEdits } = useSelect(
        ( select ) => ( {
            page: select( coreDataStore ).getEditedEntityRecord( 'postType' ),
            lastError: select( coreDataStore ).getLastEntitySaveError( 'postType' ),
            isSaving: select( coreDataStore ).isSavingEntityRecord( 'postType' ),
            hasEdits: select( coreDataStore ).hasEditsForEntityRecord( 'postType' )
        } ),
        [pageId]
    );

    const { saveEditedEntityRecord, editEntityRecord } = useDispatch( coreDataStore );
    const handleSave = async () => {
        const savedRecord = await saveEditedEntityRecord( 'postType', 'page' );
        if ( savedRecord ) {
            onSaveFinished();
        }
    };
    const handleChange = ( title ) => editEntityRecord( 'postType', 'page', { title } );

    return (
        <PageForm
            title={ page.title }
            onChangeTitle={ handleChange }
            hasEdits={ hasEdits }
            lastError={ lastError }
            isSaving={ isSaving }
            onCancel={ onCancel }
            onSave={ handleSave }
        />
    );
}

```

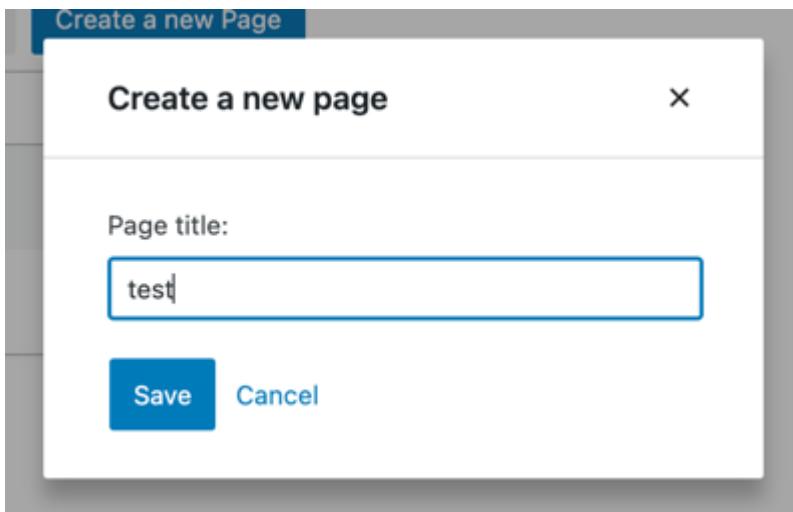
```

}

function PageForm( { title, onChangeTitle, hasEdits, lastError, isSaving,
  return (
    <div className="my-gutenberg-form">
      <TextControl
        label="Page title:"
        value={ title }
        onChange={ onChangeTitle }
      />
      { lastError ? (
        <div className="form-error">Error: { lastError.message }</div>
      ) : (
        false
      ) }
      <div className="form-buttons">
        <Button
          onClick={ onSave }
          variant="primary"
          disabled={ !hasEdits || isSaving }
        >
          { isSaving ? (
            <>
              <Spinner/>
              Saving
            </>
          ) : 'Save' }
        </Button>
        <Button
          onClick={ onCancel }
          variant="tertiary"
          disabled={ isSaving }
        >
          Cancel
        </Button>
      </div>
    </div>
  );
}

```

All that's left is to refresh the page and enjoy the form:



What's next?

- **Next part:** [Adding a delete button](#)
- **Previous part:** [Building an edit form](#)
- (optional) Review the [finished app](#) in the block-development-examples repository

First published

May 18, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Part 4: Building a Create page form”](#)

[Previous Building an edit form](#) [Previous: Building an edit form](#)
[Next Adding a delete button](#) [Next: Adding a delete button](#)

Adding a delete button

In this article

Table of Contents

- [Step 1: Add a Delete button](#)
 - [Step 2: Wire the button to a delete action](#)
 - [Step 3: Add visual feedback](#)
 - [Step 4: Handle errors](#)
 - [Wiring it all together](#)
- [What's next?](#)

[↑ Back to top](#)

In the [previous part](#) we added an ability to create new pages, and in this part we will add a *Delete* feature to our app.

Here's a glimpse of what we're going to build:

Pages	
Search	
Title	Actions
Home	<button>Edit</button> <button>Delete</button>
About us	<button>Edit</button> <button>Delete</button>
Contact	<button>Edit</button> <button>Delete</button>

Step 1: Add a Delete button

Let's start by creating the `DeletePageButton` component and updating the user interface of our `PagesList` component:

```
import { Button } from '@wordpress/components';
import { decodeEntities } from '@wordpress/html-entities';

const DeletePageButton = () => (
  <Button variant="primary">
    Delete
  </Button>
)

function PagesList( { hasResolved, pages } ) {
  if ( ! hasResolved ) {
    return <Spinner />;
  }
  if ( ! pages?.length ) {
    return <div>No results</div>;
  }

  return (
    <table className="wp-list-table widefat fixed striped table-view-l
    <thead>
      <tr>
        <td>Title</td>
        <td style={{width: 190}}>Actions</td>
```

```

        </tr>
    </thead>
    <tbody>
        { pages?.map( ( page ) => (
            <tr key={page.id}>
                <td>{ decodeEntities( page.title.rendered ) }</td>
                <td>
                    <div className="form-buttons">
                        <PageEditButton pageId={ page.id } />
                        {/* ↓ This is the only change in the PagesList component */}
                        <DeletePageButton pageId={ page.id } />
                    </div>
                </td>
            </tr>
        ) ) }
    </tbody>
</table>
);
}

```

This is what the PagesList should look like now:

Pages	
Search	Actions
Home	Edit Delete
About us	Edit Delete
Contact	Edit Delete

Step 2: Wire the button to a delete action

In Gutenberg data, we delete entity records from the WordPress REST API using the `deleteEntityRecord` action. It sends the request, processes the result, and updates the cached data in the Redux state.

Here's how you can try deleting entity records in your browser's dev tools:

```
// We need a valid page ID to call deleteEntityRecord, so let's get the first page
const pageId = wp.data.select( 'core' ).getEntityRecords( 'postType', 'page' ).map( ( page ) => page.id )[ 0 ]
```

```
// Now let's delete that page:  
const promise = wp.data.dispatch( 'core' ).deleteEntityRecord( 'postType',  
  
// promise gets resolved or rejected when the API request succeeds or fails
```

Once the REST API request is finished, you will notice one of the pages has disappeared from the list. This is because that list is populated by the `useSelect()` hook and the `select(coreDataStore).getEntityRecords('postType', 'page')` selector. Anytime the underlying data changes, the list gets re-rendered with fresh data. That's pretty convenient!

Let's dispatch that action when `DeletePageButton` is clicked:

```
const DeletePageButton = ({ pageId }) => {  
  const { deleteEntityRecord } = useDispatch( coreDataStore );  
  const handleDelete = () => deleteEntityRecord( 'postType', 'page', pageId );  
  return (  
    <Button variant="primary" onClick={ handleDelete }>  
      Delete  
    </Button>  
  );  
}
```

Step 3: Add visual feedback

It may take a few moments for the REST API request to finish after clicking the `Delete` button. Let's communicate that with a `<Spinner />` component similarly to what we did in the previous parts of this tutorial.

We'll need the `isDeletingEntityRecord` selector for that. It is similar to the `isSavingEntityRecord` selector we've already seen in [part 3](#): it returns `true` or `false` and never issues any HTTP requests:

```
const DeletePageButton = ({ pageId }) => {  
  // ...  
  const { isDeleting } = useSelect(  
    select => ({  
      isDeleting: select( coreDataStore ).isDeletingEntityRecord( 'postType', pageId )  
    })  
  );  
  return (  
    <Button variant="primary" onClick={ handleDelete } disabled={ isDeleting }>  
      { isDeleting ? (  
        <>  
        <Spinner />  
        Deleting...  
      ) : 'Delete' }  
    </Button>  
  );  
}
```

Here's what it looks like in action:

Pages

Search		Actions
Title		
Home	<button>Edit</button>	<button>Deleting...</button>
About us	<button>Edit</button>	<button>Delete</button>
Contact	<button>Edit</button>	<button>Delete</button>

Step 4: Handle errors

We optimistically assumed that a *delete* operation would always succeed. Unfortunately, under the hood, it is a REST API request that can fail in many ways:

- The website can be down.
- The delete request may be invalid.
- The page could have been deleted by someone else in the meantime.

To tell the user when any of these errors happen, we need to extract the error information using the `getLastEntityDeleteError` selector:

```
// Replace 9 with an actual page ID
wp.data.select('core').getLastEntityDeleteError('postType', 'page', 9)
```

Here's how we can apply it in `DeletePageButton`:

```
import { useEffect } from 'react';
const DeletePageButton = ({ pageId }) => {
    // ...
    const { error, /* ... */ } = useSelect(
        select => ( {
            error: select( coreDataStore ).getLastEntityDeleteError( 'postType',
                'page', pageId )
        }
    );
    useEffect( () => {
        if ( error ) {
            // Display the error
        }
    }, [error] )
}
```

```
// ...
}
```

The `error` object comes from the `@wordpress/api-fetch` and contains information about the error. It has the following properties:

- `message` – a human-readable error message such as `Invalid post ID`.
- `code` – a string-based error code such as `rest_post_invalid_id`. To learn about all possible error codes you'd need to refer to the [/v2/pages endpoint's source code](#).
- `data` (optional) – error details, contains the `code` property containing the HTTP response code for the failed request.

There are many ways to turn that object into an error message, but in this tutorial, we will display the `error.message`.

WordPress has an established pattern of displaying status information using the **Snackbar** component. Here's what it looks like **in the Widgets editor**:



core



0



New

Dashboard

My first Gutenberg app

Posts

Media

Pages

Comments

Appearance

Themes

Customize

Widgets

Menus

Header

Background

Theme File Editor

Plugins

Users

Tools

Settings

Widgets



Sidebar

I am a widget!

Footer Widgets

Footer Widgets

Footer Widgets

Inactive widgets

Widgets saved.

Let's use the same type of notifications in our plugin! There are two parts to this:

1. Displaying notifications
2. Dispatching notifications

Displaying notifications

Our application only knows how to display pages but does not know how to display notifications. Let's tell it!

WordPress conveniently provides us with all the React components we need to render notifications. A [component called Snackbar](#) represents a single notification:

Use Snackbars to communicate low priority, non-interacting messages.

We won't use `Snackbar` directly, though. We'll use the `SnackbarList` component, which can display multiple notices using smooth animations and automatically hide them after a few seconds. In fact, WordPress uses the same component used in the Widgets editor and other wp-admin pages!

Let's create our own `Notifications` components:

```
import { SnackbarList } from '@wordpress/components';
import { store as noticesStore } from '@wordpress/notices';

function Notifications() {
    const notices = []; // We'll come back here in a second!

    return (
        <SnackbarList
            notices={ notices }
            className="components-editor-notices__snackbar"
        />
    );
}
```

The basic structure is in place, but the list of notifications it renders is empty. How do we populate it? We'll lean on the same package as WordPress: [@wordpress/notices](#).

Here's how:

```
import { SnackbarList } from '@wordpress/components';
import { store as noticesStore } from '@wordpress/notices';

function Notifications() {
    const notices = useSelect(
```

```

        ( select ) => select( noticesStore ).getNotices(),
        []
    );
const { removeNotice } = useDispatch( noticesStore );
const snackbars = notices.filter( ({ type }) => type === 'snackbar' );

return (
    <SnackbarList
        notices={ snackbars }
        className="components-editor-notices__snackbar"
        onRemove={ removeNotice }
    />
);
}

function MyFirstApp() {
    // ...
    return (
        <div>
            {/* ... */}
            <Notifications />
        </div>
    );
}

```

This tutorial is focused on managing the pages and won't discuss the above snippet in detail. If you're interested in the details of `@wordpress/notices`, the [handbook page](#) is a good place to start.

Now we're ready to tell the user about any errors that may have occurred.

Dispatching notifications

With the SnackbarNotices component in place, we're ready to dispatch some notifications! Here's how:

```

import { useEffect } from 'react';
import { store as noticesStore } from '@wordpress/notices';
function DeletePageButton( { pageId } ) {
    const { createSuccessNotice, createErrorNotice } = useDispatch( noticesStore );
    // useSelect returns a list of selectors if you pass the store handle
    // instead of a callback:
    const { getLastEntityDeleteError } = useSelect( coreDataStore );
    const handleDelete = async () => {
        const success = await deleteEntityRecord( 'postType', 'page', pageId );
        if ( success ) {
            // Tell the user the operation succeeded:
            createSuccessNotice( "The page was deleted!", {
                type: 'snackbar',
            } );
        } else {
            // We use the selector directly to get the fresh error *after*
            // have failed.
            const lastError = getLastEntityDeleteError( 'postType', 'page' );
        }
    };
}

```

```
        const message = ( lastError?.message || 'There was an error.' );
        // Tell the user how exactly the operation has failed:
        createErrorNotice( message, {
            type: 'snackbar',
        } );
    }
}
// ...
```

Great! `DeletePageButton` is now fully aware of errors. Let's see that error message in action. We'll trigger an invalid delete and let it fail. One way to do this is to multiply the `pageId` by a large number:

```
function DeletePageButton( { pageId, onCancel, onSaveFinished } ) {
    pageId = pageId * 1000;
    // ...
}
```

Once you refresh the page and click any `Delete` button, you should see the following error message:

Pages

Search	Actions
Title	Actions
Page with buttons	Edit Delete
Home1	Edit Delete
Contact	Edit Delete

Invalid post ID. Please refresh the page and try again.

Fantastic! We can now **remove the pageId = pageId * 1000; line**.

Let's now try actually deleting a page. Here's what you should see after refreshing your browser and clicking the Delete button:

Pages

Search	Actions
Title	
Home1	<button>Edit</button> <button>Delete</button>
Contact	<button>Edit</button> <button>Delete</button>

The page was deleted!

And that's it!

Wiring it all together

All the pieces are in place, great! Here's all the changes we've made in this chapter:

```
import { useState, useEffect } from 'react';
import { useDispatch, useSelector } from '@wordpress/data';
import { Button, Modal, TextControl } from '@wordpress/components';

function MyFirstApp() {
    const [searchTerm, setSearchTerm] = useState( '' );
    const { pages, hasResolved } = useSelector(
        ( select ) => {
            const query = {};
            if ( searchTerm ) {
                query.search = searchTerm;
            }
            const selectorArgs = [ 'postType', 'page', query ];
            const pages = select( coreDataStore ).getEntityRecords( ...sel
    return {
        pages,
        hasResolved: select( coreDataStore ).hasFinishedResolution(
            'getEntityRecords',
```

```
        selectorArgs,
    ),
},
[searchTerm],
);

return (
<div>
    <div className="list-controls">
        <SearchControl onChange={ setSearchTerm } value={ searchTerm } />
        <PageCreateButton/>
    </div>
    <PagesList hasResolved={ hasResolved } pages={ pages } />
    <Notifications />
</div>
);
}

function SnackbarNotices() {
    const notices = useSelect(
        ( select ) => select( noticesStore ).getNotices(),
        []
    );
    const { removeNotice } = useDispatch( noticesStore );
    const snackbarNotices = notices.filter( ( { type } ) => type === 'snac

    return (
        <SnackbarList
            notices={ snackbarNotices }
            className="components-editor-notices__snackbar"
            onRemove={ removeNotice }
        />
    );
}

function PagesList( { hasResolved, pages } ) {
    if ( !hasResolved ) {
        return <Spinner/>;
    }
    if ( !pages?.length ) {
        return <div>No results</div>;
    }

    return (
        <table className="wp-list-table widefat fixed striped table-view-l
            <thead>
                <tr>
                    <td>Title</td>
                    <td style={ { width: 190 } }>Actions</td>
                </tr>
            </thead>
            <tbody>
                { pages?.map( ( page ) => (

```

```

        <tr key={ page.id }>
            <td>{ page.title.rendered }</td>
            <td>
                <div className="form-buttons">
                    <PageEditButton pageId={ page.id }/>
                    <DeletePageButton pageId={ page.id }/>
                </div>
            </td>
        </tr>
    ) ) )
</tbody>
</table>
);
}

function DeletePageButton( { pageId } ) {
    const { createSuccessNotice, createErrorNotice } = useDispatch( notice );
    // useSelect returns a list of selectors if you pass the store handle
    // instead of a callback:
    const { getLastEntityDeleteError } = useSelect( coreDataStore )
    const handleDelete = async () => {
        const success = await deleteEntityRecord( 'postType', 'page', pageId );
        if ( success ) {
            // Tell the user the operation succeeded:
            createSuccessNotice( "The page was deleted!", {
                type: 'snackbar',
            } );
        } else {
            // We use the selector directly to get the error at this point
            // Imagine we fetched the error like this:
            //   const { lastError } = useSelect( function() { /* ... */ }
            // Then, lastError would be null inside of handleDelete.
            // Why? Because we'd refer to the version of it that was computed
            // before the handleDelete was even called.
            const lastError = getLastEntityDeleteError( 'postType', 'page' );
            const message = ( lastError?.message || 'There was an error.' );
            // Tell the user how exactly the operation have failed:
            createErrorNotice( message, {
                type: 'snackbar',
            } );
        }
    }
}

const { deleteEntityRecord } = useDispatch( coreDataStore );
const { isDeleting } = useSelect(
    select => ( {
        isDeleting: select( coreDataStore ).isDeletingEntityRecord( 'page' ),
    } ),
    [ pageId ]
);

return (
    <Button variant="primary" onClick={ handleDelete } disabled={ isDeleting ? (

```

```
        <>
        <Spinner />
        Deleting...
      </>
    ) : 'Delete' }
</Button>
);
}
```

What's next?

- Previous part: [Building a Create page form](#)
- (optional) Review the [finished app](#) in the block-development-examples repository

First published

July 1, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Adding a delete button”](#)

[Previous Part 4: Building a Create page form](#) [Previous: Part 4: Building a Create page form](#)
[Next Curating the Editor Experience](#) [Next: Curating the Editor Experience](#)

Curating the Editor Experience

In this article

Table of Contents

- [Combining approaches](#)
- [Additional resources](#)

[↑ Back to top](#)

Curating the editing experience in WordPress is important because it allows you to streamline the editing process, ensuring consistency and alignment with the site’s style and branding guidelines. It also makes it easier for users to create and manage content effectively without accidental modifications or layout changes. This leads to a more efficient and personalized experience.

The purpose of this guide is to offer various ways you can lock down and curate the experience of using WordPress, especially with the introduction of more design tools and the Site Editor.

In this section, you will learn:

1. [Block locking](#): how to restrict user interactions with specific blocks in the Editor for better content control
2. [Patterns](#): about creating and implementing predefined block layouts to ensure design and content uniformity
3. [theme.json](#): to configure global styles and settings for your theme using the theme.json file
4. [Filters and hooks](#): about the essential filters and hooks used to modify the Editor
5. [Disabling Editor functionality](#): about additional ways to selectively disable features or components in the Editor to streamline the user experience

[Combining approaches](#)

Remember that the approaches provided in the documentation above can be combined as you see fit. For example, you can provide custom patterns to use when creating a new page while also limiting the amount of customization that can be done to aspects of them, like only allowing specific preset colors to be used for the background of a Cover block or locking down what blocks can be deleted.

When considering the approaches to take, think about the specific ways you might want to both open up the experience and curate it.

[Additional resources](#)

- [Builder Basics – Working with Templates in Full Site Editing \(Part 3\)](#)
- [Core Editor Improvement: Curated experiences with locking APIs & theme.json](#)
- [Learn WordPress session on Curating the Editor Experience](#)

First published

July 4, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Curating the Editor Experience”](#)

[Previous Adding a delete button](#) [Previous: Adding a delete button](#)

[Next Block Locking API](#) [Next: Block Locking API](#)

Block Locking API

In this article

Table of Contents

- [Lock the ability to move or remove specific blocks](#)
- [Lock the ability to edit certain blocks](#)

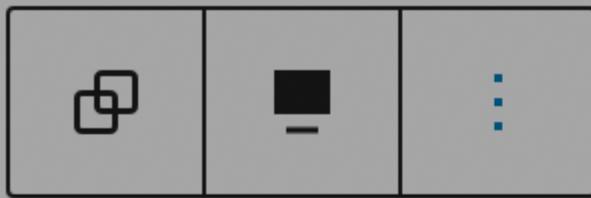
- [Apply block locking to patterns or templates](#)
- [Apply content-only editing in patterns or templates](#)
- [Change permissions to control locking ability](#)

[↑ Back to top](#)

The Block Locking API allows you to restrict actions on specific blocks within the Editor. This API can be used to prevent users from moving, removing, or editing certain blocks, ensuring layout consistency and content integrity.

Lock the ability to move or remove specific blocks

Users can lock and unlock blocks via the Editor. The locking UI has options for preventing blocks from being moved within the content canvas or removed:



My Sup

Hide more settings ⌘,

Copy block

Duplicate ⌘D

Insert before ⌘T

Insert after ⌘Y

Lock 🔒

Add to Reusable blocks ◆

Group

Ungroup

Remove Group ⌘Z

Jen

Pe

Jen

JENSEN, DUSTON DE

Keep in mind that you can apply locking options to blocks nested inside of a containing block by turning on the “Apply to all blocks inside” option. However, you cannot mass lock blocks otherwise.

Lock the ability to edit certain blocks

Alongside the ability to lock moving or removing blocks, the [Navigation Block](#) and [Reusable block](#) have an additional capability: lock the ability to edit the contents of the block. This locks the ability to make changes to any blocks inside of either block type.

Apply block locking to patterns or templates

When building patterns or templates, theme authors can use these same UI tools to set the default locked state of blocks. For example, a theme author could lock various pieces of a header. Keep in mind that by default, users with editing access can unlock these blocks. [Here's an example of a pattern](#) with various blocks locked in different ways and here's more context on [creating a template with locked blocks](#). You can build these patterns in the Editor itself, including adding locking options, before following the [documentation to register them](#).

Apply content-only editing in patterns or templates

This functionality was introduced in WordPress 6.1. In contrast to block locking, which disables the ability to move or remove blocks, content-only editing is both designed for use at the pattern or template level and hides all design tools, while still allowing for the ability to edit the content of the blocks. This provides a great way to simplify the interface for users and preserve a design. When this option is added, the following changes occur:

- Non-content child blocks (containers, spacers, columns, etc) are hidden from list view, unclickable on the canvas, and entirely un-editable.
- The Inspector will display a list of all child ‘content’ blocks. Clicking a block in this list reveals its settings panel.
- The main List View only shows content blocks, all at the same level regardless of actual nesting.
- Children blocks within the overall content locked container are automatically move / remove locked.
- Additional child blocks cannot be inserted, further preserving the design and layout.
- There is a link in the block toolbar to ‘Modify’ that a user can toggle on/off to have access to the broader design tools. Currently, it's not possible to programmatically remove this option.

This option can be applied to Columns, Cover, and Group blocks as well as third-party blocks that have the templateLock attribute in its block.json. To adopt this functionality, you need to use "templateLock": "contentOnly". [Here's an example of a pattern](#) with this functionality in place. For more information, please [review the relevant documentation](#).

Note: There is no UI in place to manage content locking and it must be managed at the code level.

Change permissions to control locking ability

Agencies and plugin authors can offer an even more curated experience by limiting which users have [permission to lock and unlock blocks](#). By default, anyone who is an administrator will have access to lock and unlock blocks.

Developers can add a filter to the [block_editor_settings_all](#) hook to configure permissions around locking blocks. The hook passes two parameters to the callback function:

- `$settings` – An array of configurable settings for the Editor.
- `$context` – An instance of [WP_Block_Editor_Context](#), an object that contains information about the current Editor.

Specifically, developers can alter the `$settings['canLockBlocks']` value by setting it to `true` or `false`, typically by running through one or more conditional checks.

The following example disables block locking permissions for all users when editing a page:

```
add_filter( 'block_editor_settings_all', function( $settings, $context ) {  
    if ( $context->post && 'page' === $context->post->post_type ) {  
        $settings['canLockBlocks'] = false;  
    }  
  
    return $settings;  
}, 10, 2 );
```

Another common use case may be to only allow users who can edit the visual design of the site (theme editing) to lock or unlock blocks. Now, the best option would be to test against the `edit_theme_options` capability, as shown in the following code snippet:

```
add_filter( 'block_editor_settings_all', function( $settings ) {  
    $settings['canLockBlocks'] = current_user_can( 'edit_theme_options' );  
  
    return $settings;  
} );
```

Developers may use any type of conditional check to determine who can lock/unlock blocks. This is merely a small sampling of what is possible via the filter hook.

First published

December 27, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Block Locking API](#)

[Previous Curating the Editor Experience](#) [Previous: Curating the Editor Experience](#)
[Next Patterns](#) [Next: Patterns](#)

Patterns

In this article

Table of Contents

- [Prioritize starter patterns for any post type](#)
- [Prioritize starter patterns for template creation](#)
- [Lock patterns](#)
- [Prioritize specific patterns from the Pattern Directory](#)
- [Additional resources](#)

[↑ Back to top](#)

Block [patterns](#) are one of the best ways to provide users with unique and curated editing experiences.

[Prioritize starter patterns for any post type](#)

When a user creates new content, regardless of post type, they are met with an empty canvas. However, that experience can be improved thanks to the option to have patterns from a specific type prioritized upon creation of a new piece of content. The modal appears each time the user creates a new item when there are patterns on their website that declare support for the `core/post-content` block types. By default, WordPress does not include any of these patterns, so the modal will not appear without at least two of these post content patterns being added.

To opt into this, include `core/post-content` in the Block Types for your pattern. From there, you can control which post types the pattern should show up for via the Post Types option. Here's an example of a pattern that would appear when creating a new post.

```
<?php
/**
 * Title: New Event Announcement
 * Slug: twentytwentytwo/new-event-announcement
 * Block Types: core/post-content
 * Post Types: post
 * Categories: featured, text
 */
?>

<!-- wp:heading {"lock":{"move":false,"remove":true}} -->
<h2>Details</h2>
<!-- /wp:heading -->

<!-- wp:heading {"lock":{"move":false,"remove":true}} -->
<h2>Directions</h2>
<!-- /wp:heading -->

<!-- wp:heading {"lock":{"move":false,"remove":true}} -->
<h2>RSVP</h2>
```

```

<!-- /wp:heading -->

<!-- wp:paragraph {"lock":{"move":true,"remove":true}} -->
<p>To RSVP, please join the #fse-outreach-experiment in Make Slack. </p>
<!-- /wp:paragraph -->

<!-- wp:buttons -->
<div class="wp-block-buttons"><!-- wp:button {"lock":{"move":true,"remove":true}} -->
<div class="wp-block-button"><a class="wp-block-button__link wp-element-button" href="#">Join</a>
<!-- /wp:button --></div>
<!-- /wp:buttons -->

<!-- wp:cover {"useFeaturedImage":true,"dimRatio":80,"overlayColor":"primary","lock":{"move":true,"remove":true}} -->
<div class="wp-block-cover alignfull"><span aria-hidden="true" class="wp-block-cover__image-background" style="background-image: url(https://make.org/wp-content/themes/make/assets/images/placeholder-image.png);"></span>
<p class="has-text-align-center has-large-font-size">We hope to see you there!
<!-- /wp:paragraph --></div></div>
<!-- /wp:cover -->

```

Read more about this functionality in the [Page creation patterns in WordPress 6.0 dev note](#) and [note that WordPress 6.1 brought this functionality to all post types](#).

Prioritize starter patterns for template creation

In the same way patterns can be prioritized for new posts or pages, the same experience can be added to the template creation process. When patterns declare support for the ‘templateTypes’ property, the patterns will appear anytime a template that matches the designation is created, along with the options to start from a blank state or use the current fallback of the template. By default, WordPress does not include any of these patterns.

To opt into this, a pattern needs to specify a property called `templateTypes`, which is an array containing the templates where the patterns can be used as the full content. Here’s an example of a pattern that would appear when creating a 404 template:

```

register_block_pattern(
    'wp-my-theme/404-template-pattern',
    array(
        'title'      => ___( '404 Only template pattern', 'wp-my-theme' ),
        'templateTypes' => array( '404' ),
        'content'     => '<!-- wp:paragraph {"align":"center","fontSize":16,"fontWeight":"bold"} -->
<p>404</p>
<!-- /wp:paragraph -->',
    )
);

```

Read more about this functionality in the [Patterns on the create a new template modal in the WordPress 6.3 dev note](#).

Lock patterns

As mentioned in the prior section on Locking APIs, aspects of patterns themselves can be locked so that the important aspects of the design can be preserved. [Here’s an example of a pattern](#) with various blocks locked in different ways. You can build these patterns in the editor itself, including adding locking options, before [following the documentation to register them](#).

Prioritize specific patterns from the Pattern Directory

With WordPress 6.0 themes can register patterns from [Pattern Directory](#) through theme.json. To accomplish this, themes should use the new patterns top level key in theme.json. Within this field, themes can list patterns to register from the Pattern Directory. The patterns field is an array of pattern slugs from the Pattern Directory. Pattern slugs can be extracted by the url in a single pattern view at the Pattern Directory. Example: This url <https://wordpress.org/patterns/partner-logos> the slug is partner-logos.

```
{  
    "version": 2,  
    "patterns": [ "short-text-surrounded-by-round-images", "partner-logos"  
}
```

Note that this field requires using [version 2 of theme.json](#). The content creator will then find the respective Pattern in the inserter “Patterns” tab in the categories that match the categories from the Pattern Directory.

Additional resources

- [Using template patterns to build multiple homepage designs](#) (WordPress Developer Blog)

First published

December 27, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Patterns”](#)

[Previous Block Locking API](#) [Previous: Block Locking API](#)
[Next theme.json](#) [Next: theme.json](#)

theme.json

In this article

Table of Contents

- [Providing default controls/options](#)
- [Limiting interface options with theme.json](#)
 - [Limit options on a per-block basis](#)
 - [Disable inherit default layout](#)
 - [Limit options globally](#)

[↑ Back to top](#)

A theme's theme.json file is one of the best ways to curate the Editor experience and will likely be the first tool you use before reaching for more sophisticated solutions.

Providing default controls/options

Since theme.json acts as a configuration tool, there are numerous ways to define at a granular level what options are available. This section will use duotone as an example since it showcases a feature that cuts across a few blocks and allows for varying levels of access.

Duotone with Core options and customization available for each image related block:

```
{  
  "version": 2,  
  "settings": {  
    "color": {  
      "customDuotone": true,  
      "duotone": [  
        ]  
    }  
  }  
}
```

Duotone with theme defined color options, Core options, and customization available for each image related block:

```
{  
  "version": 2,  
  "settings": {  
    "color": {  
      "duotone": [  
        {  
          "colors": [ "#000000", "#ffffff" ],  
          "slug": "foreground-and-background",  
          "name": "Foreground and background"  
        },  
        {  
          "colors": [ "#000000", "#ff0200" ],  
          "slug": "foreground-and-secondary",  
          "name": "Foreground and secondary"  
        },  
        {  
          "colors": [ "#000000", "#7f5dee" ],  
          "slug": "foreground-and-tertiary",  
          "name": "Foreground and tertiary"  
        },  
        ]  
      }  
    }  
  }  
}
```

Duotone with defined default options and all customization available for the Post Featured Image block:

```
{
  "schema": "https://schemas.wp.org/trunk/theme.json",
  "version": 2,
  "settings": {
    "color": {
      "custom": true,
      "customDuotone": true
    },
    "blocks": {
      "core/post-featured-image": {
        "color": {
          "duotone": [
            {
              "colors": [ "#282828", "#ff5837" ],
              "slug": "black-and-orange",
              "name": "Black and Orange"
            },
            {
              "colors": [ "#282828", "#0288d1" ],
              "slug": "black-and-blue",
              "name": "Black and Blue"
            }
          ],
          "customDuotone": true,
          "custom": true
        }
      }
    }
  }
}
```

Duotone with only defined default options and core options available for the Post Featured Image block (no customization):

```
{
  "schema": "https://schemas.wp.org/trunk/theme.json",
  "version": 2,
  "settings": {
    "color": {
      "custom": true,
      "customDuotone": true
    },
    "blocks": {
      "core/post-featured-image": {
        "color": {
          "duotone": [
            {
              "colors": [ "#282828", "#ff5837" ],
              "slug": "black-and-orange",
              "name": "Black and Orange"
            },
            {
              "colors": [ "#282828", "#0288d1" ],
              "slug": "black-and-blue",
              "name": "Black and Blue"
            }
          ]
        }
      }
    }
  }
}
```

```
        "name": "Black and Blue"
    }
],
"customDuotone": false,
"custom": false
}
}
}
}
```

[Limiting interface options with theme.json](#)

[Limit options on a per-block basis](#)

Beyond defining default values, using theme.json allows you to also remove options entirely and instead rely on what the theme has set in place. Below is a visual showing two extremes with the same paragraph block:

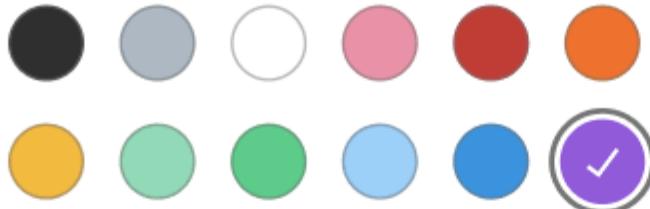
Vivid purple

9B51E0

THEME



DEFAULT



Unlocked

Continuing the examples with duotone, this means you could allow full access to all Duotone functionality for Image blocks and only limit the Post Featured Image block like so:

```
{  
    "schema": "https://schemas.wp.org/trunk/theme.json",  
    "version": 2,  
    "settings": {  
        "color": {  
            "custom": true,  
            "customDuotone": true  
        },  
        "blocks": {  
            "core/image": {  
                "color": {  
                    "duotone": [],  
                    "customDuotone": true,  
                    "custom": true  
                }  
            },  
            "core/post-featured-image": {  
                "color": {  
                    "duotone": [],  
                    "customDuotone": false,  
                    "custom": false  
                }  
            }  
        }  
    }  
}
```

You can read more about how best to [turn on/off options with theme.json here](#).

Disable inherit default layout

To disable the “Inherit default layout” setting for container blocks like the Group block, remove the following section:

```
"layout": {  
    "contentSize": null,  
    "wideSize": null  
},
```

Limit options globally

When using theme.json in a block or classic theme, these settings will stop the default color and typography controls from being enabled globally, greatly limiting what's possible:

```
{  
    "$schema": "http://schemas.wp.org/trunk/theme.json",  
    "version": 2,  
    "settings": {  
        "layout": {  
            "contentSize": "750px"  
        },  
    }  
}
```

```
        "color": {
            "background": false,
            "custom": false,
            "customDuotone": false,
            "customGradient": false,
            "defaultGradients": false,
            "defaultPalette": false,
            "text": false
        },
        "typography": {
            "customFontSize": false,
            "dropCap": false,
            "fontStyle": false,
            "fontWeight": false,
            "letterSpacing": false,
            "lineHeight": false,
            "textDecoration": false,
            "textTransform": false
        }
    }
}
```

To enable something from the above, just set whatever value you want to change to `true` for more granularity.

First published

April 26, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: theme.json](#)

[Previous Patterns](#) [Previous: Patterns](#)
[Next Filters and hooks](#) [Next: Filters and hooks](#)

Filters and hooks

In this article

[Table of Contents](#)

- [Server-side theme.json filters](#)
- [Client-side \(Editor\) filters](#)
- [Additional resources](#)

[↑ Back to top](#)

The Editor provides numerous filters and hooks that allow you to modify the editing experience. Here are a few.

Server-side theme.json filters

The theme.json file is a great way to control interface options, but it only allows for global or block-level modifications, which can be limiting in some scenarios.

For instance, in the previous section, color and typography controls were disabled globally using theme.json. But let's say you want to enable color settings for users who are Administrators.

To provide more flexibility, WordPress 6.1 introduced server-side filters allowing you to customize theme.json data at four different data layers.

- [wp_theme_json_data_default](#) – Hooks into the default data provided by WordPress
- [wp_theme_json_data_blocks](#) – Hooks into the data provided by blocks.
- [wp_theme_json_data_theme](#) – Hooks into the data provided by the current theme.
- [wp_theme_json_data_user](#) – Hooks into the data provided by the user.

In the following example, the data from the current theme's theme.json file is updated using the [wp_theme_json_data_theme](#) filter. Color controls are restored if the current user is an Administrator.

```
// Disable color controls for all users except Administrators.
function example_filter_theme_json_data_theme( $theme_json ){
    $is_administrator = current_user_can( 'edit_theme_options' );

    if ( $is_administrator ) {
        $new_data = array(
            'version' => 2,
            'settings' => array(
                'color' => array(
                    'background' => true,
                    'custom' => true,
                    'customDuotone' => true,
                    'customGradient' => true,
                    'defaultGradients' => true,
                    'defaultPalette' => true,
                    'text' => true,
                ),
            ),
        );
    }

    return $theme_json->update_with( $new_data );
}
add_filter( 'wp_theme_json_data_theme', 'example_filter_theme_json_data_th
```

The filter receives an instance of the `WP_Theme_JSON_Data` class with the data for the respective layer. Then, you pass new data in a valid theme.json-like structure to the `update_with($new_data)` method. A theme.json version number is required in `$new_data`.

Client-side (Editor) filters

WordPress 6.2 introduced a new client-side filter allowing you to modify block-level [theme.json settings](#) before the Editor is rendered.

The filter is called `blockEditor.useSetting.before` and can be used in the JavaScript code as follows:

```
import { addFilter } from '@wordpress/hooks';

/**
 * Limit the Column block's spacing options to pixels.
 */
addFilter(
    'blockEditor.useSetting.before',
    'example/useSetting.before',
    ( settingValue, settingName, clientId, blockName ) => {
        if ( blockName === 'core/column' && settingName === 'spacing.units' ) {
            return [ 'px' ];
        }
        return settingValue;
    }
);

```

This example will restrict the available spacing units for the Column block to just pixels. As discussed above, a similar restriction could be applied using theme.json filters or directly in a theme's theme.json file using block-level settings.

However, the `blockEditor.useSetting.before` filter is unique because it allows you to modify settings according to the block's location, neighboring blocks, the current user's role, and more. The possibilities for customization are extensive.

In the following example, text color controls are disabled for the Heading block whenever the block is placed inside of a Media & Text block.

```
import { select } from '@wordpress/data';
import { addFilter } from '@wordpress/hooks';

/**
 * Disable text color controls on Heading blocks when placed inside of Media & Text blocks.
 */
addFilter(
    'blockEditor.useSetting.before',
    'example/useSetting.before',
    ( settingValue, settingName, clientId, blockName ) => {
        if ( blockName === 'core/heading' ) {
            const { getBlockParents, getBlockName } = select( 'core/block-editor' );
            const blockParents = getBlockParents( clientId, true );
            const inMediaText = blockParents.some( ( ancestorId ) => getBlockName(
                ancestorId
            ) === 'core/media-text' );
            if ( inMediaText && settingName === 'color.text' ) {
                return false;
            }
        }
    }
);

```

```
        return settingValue;
    }
);
```

Additional resources

- [How to modify theme.json data using server-side filters](#) (WordPress Developer Blog)
- [Curating the Editor experience with client-side filters](#) (WordPress Developer Blog)

First published

December 27, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Filters and hooks”](#)

[Previous theme.json](#) [Previous: theme.json](#)

[Next Disable Editor functionality](#) [Next: Disable Editor functionality](#)

Disable Editor functionality

In this article

Table of Contents

- [Restrict block options](#)
- [Disable the Pattern Directory](#)
- [Disable block variations](#)
- [Disable block styles](#)
- [Disable access to the Template Editor](#)
- [Disable access to the Code Editor](#)

[↑ Back to top](#)

This page is dedicated to the many ways you can disable specific functionality in the Post Editor and Site Editor that are not covered in other areas of the curation documentation.

Restrict block options

There might be times when you don't want access to a block at all to be available for users. To control what's available in the inserter, you can take two approaches: [an allow list](#) that disables all blocks except those on the list or a [deny list that unregisters specific blocks](#).

Disable the Pattern Directory

To fully remove patterns bundled with WordPress core from being accessed in the Inserter, the following can be added to your `functions.php` file:

```
function example_theme_support() {
    remove_theme_support( 'core-block-patterns' );
}
add_action( 'after_setup_theme', 'example_theme_support' );
```

Disable block variations

Some Core blocks are actually [block variations](#). A great example is the Row and Stack blocks, which are actually variations of the Group block. If you want to disable these “blocks”, you actually need to disable the respective variations.

Block variations are registered using JavaScript and need to be disabled with JavaScript. The code below will disable the Row variation.

```
wp.domReady( () => {
    wp.blocks.unregisterBlockVariation( 'core/group', 'group-row' );
});
```

Assuming the code was placed in a `disable-variations.js` file located in the root of your theme folder, you can enqueue this file in the theme’s `functions.php` using the code below.

```
function example_disable_variations_script() {
    wp_enqueue_script(
        'example-disable-variations-script',
        get_template_directory_uri() . '/disable-variations.js',
        array( 'wp-dom-ready' ),
        wp_get_theme()->get( 'Version' ),
        true
    );
}
add_action( 'enqueue_block_editor_assets', 'example_disable_variations_scr
```

Disable block styles

There are a few Core blocks that include their own [block styles](#). An example is the Image block, which includes a block style for rounded images called “Rounded”. You may not want your users to round images, or you might prefer to use the border-radius control instead of the block style. Either way, it’s easy to disable any unwanted block styles.

Unlike block variations, you can register styles in either JavaScript or PHP. If a style was registered in JavaScript, it must be disabled with JavaScript. If registered using PHP, the style can be disabled with either. All Core block styles are registered in JavaScript.

So, you would use the following code to disable the “Rounded” block style for the Image block.

```
wp.domReady( () => {
    wp.blocks.unregisterBlockStyle( 'core/image', 'rounded' );
});
```

This JavaScript should be enqueued much like the block variation example above. Refer to the [block styles](#) documentation for how to register and unregister styles using PHP.

[Disable access to the Template Editor](#)

Whether you're using theme.json in a Classic or Block theme, you can add the following to your functions.php file to remove access to the Template Editor that is available when editing posts or pages:

```
function example_theme_support() {
    remove_theme_support( 'block-templates' );
}
add_action( 'after_setup_theme', 'example_theme_support' );
```

This prevents both the ability to create new block templates or edit them from within the Post Editor.

[Disable access to the Code Editor](#)

The Code Editor allows you to view the underlying block markup for a page or post. While this view is handy for experienced users, you can inadvertently break block markup by editing content. Add the following to your functions.php file to restrict access.

```
function example_restrict_code_editor_access( $settings, $context ) {
    $settings[ 'codeEditingEnabled' ] = false;

    return $settings;
}
add_filter( 'block_editor_settings_all', 'example_restrict_code_editor_acc
```

This code prevents all users from accessing the Code Editor. You could also add [capability](#) checks to disable access for specific users.

First published

December 27, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Disable Editor functionality”](#)

[Previous Filters and hooks](#) [Previous: Filters and hooks](#)

[Next Enqueueing assets in the Editor](#) [Next: Enqueueing assets in the Editor](#)

Enqueueing assets in the Editor

In this article

Table of Contents

- [The Editor versus Editor content](#)
- [Scenarios for enqueueing assets](#)
 - [Editor scripts and styles](#)
 - [Editor content scripts and styles](#)
 - [Block scripts and styles](#)
 - [Theme scripts and styles](#)
- [Backward compatibility and known issues](#)

[↑ Back to top](#)

This guide is designed to be the definitive reference for enqueueing assets (scripts and styles) in the Editor. The approaches outlined here represent the recommended practices but keep in mind that this resource will evolve as WordPress does. Updates are encouraged.

As of WordPress 6.3, the Post Editor is iframed if all registered blocks have a [Block API version 3](#) or higher and no traditional metaboxes are registered. The Site Editor has always been iframed. This guide assumes you are looking to enqueue assets for the iframed Editor, but refer to the backward compatibility section below for additional considerations.

For more information about why the Editor is iframed, please revisit the post [Blocks in an iframed \(template\) editor](#).

[The Editor versus Editor content](#)

Before enqueueing assets in the Editor, you must first identify what you are trying to target.

Do you want to add styling or JavaScript to the user-generated content (blocks) in the Editor? Or do you want to modify the Editor user interface (UI) components or interact with Editor APIs? This could include everything from creating custom block controls to registering block variations.

There are different hooks to use depending on the answers to these questions, and if you are building a block or a theme, there are additional approaches to consider. Refer to the designated sections below.

[Scenarios for enqueueing assets](#)

[Editor scripts and styles](#)

Whenever you need to enqueue assets for the Editor itself (i.e. not the user-generated content), you should use the [enqueue_block_editor_assets](#) hook coupled with the standard [wp_enqueue_script](#) and [wp_enqueue_style](#) functions.

Examples might be adding custom inspector or toolbar controls, registering block styles and variations in Javascript, registering Editor plugins, etc.

```

/**
 * Enqueue Editor assets.
 */
function example_enqueue_editor_assets() {
    wp_enqueue_script(
        'example-editor-scripts',
        plugins_url( 'editor-scripts.js', __FILE__ )
    );
    wp_enqueue_style(
        'example-editor-styles',
        plugins_url( 'editor-styles.css', __FILE__ )
    );
}
add_action( 'enqueue_block_editor_assets', 'example_enqueue_editor_assets' );

```

While not the recommended approach, it's important to note that `enqueue_block_editor_assets` can be used to style Editor content for backward compatibility. See below for more details.

[Editor content scripts and styles](#)

As of WordPress 6.3, all assets added through the [`enqueue_block_assets`](#) PHP action will also be enqueued in the iframed Editor. See [#48286](#) for more details.

This is the primary method you should use to enqueue assets for user-generated content (blocks), and this hook fires both in the Editor and on the front end of your site. It should not be used to add assets intended for the Editor UI or to interact with Editor APIs. See below for a note on backward compatibility.

There are instances where you may only want to add assets in the Editor and not on the front end. You can achieve this by using an [`is_admin\(\)`](#) check.

```

/**
 * Enqueue content assets but only in the Editor.
 */
function example_enqueue_editor_content_assets() {
    if ( is_admin() ) {
        wp_enqueue_script(
            'example-editor-content-scripts',
            plugins_url( 'constant-scripts.css', __FILE__ )
        );
        wp_enqueue_style(
            'example-editor-content-styles',
            plugins_url( 'constant-styles.css', __FILE__ )
        );
    }
}
add_action( 'enqueue_block_assets', 'example_enqueue_editor_content_assets' );

```

You can also use the hook [`block_editor_settings_all`](#) to modify Editor settings directly. This method is a bit more complicated to implement but provides greater flexibility. It should only be used if `enqueue_block_assets` does not meet your needs.

The following example sets the default text color for all paragraphs to green.

```

/**
 * Modify the Editor settings by adding custom styles.
 *
 * @param array $editor_settings An array containing the current Editor settings.
 * @param string $editor_context The context of the editor.
 *
 * @return array Modified editor settings with the added custom CSS style.
 */
function example_modify_editor_settings( $editor_settings, $editor_context ) {
    $editor_settings["styles"][] = array(
        "css" => 'p { color: green }'
    );

    return $editor_settings;
}
add_filter( 'block_editor_settings_all', 'example_modify_editor_settings' );

```

These styles are inlined in the body of the iframed Editor and prefixed by `.editor-styles-wrapper`. The resulting markup will look like this:

```
<style>.editor-styles-wrapper p { color: green; }</style>
```

Beginning in WordPress 6.3, you can also use this method of modifying Editor settings to change styles dynamically with JavaScript. See [#52767](#) for more details.

[Block scripts and styles](#)

When building a block, `block.json` is the recommended way to enqueue all scripts and styles that are specifically required for the block itself. You are able to enqueue assets for the Editor, the front end, or both. See the [Block Metadata](#) article for more details.

[Theme scripts and styles](#)

If you need to enqueue Editor JavaScript in a theme, you can use either `enqueue_block_assets` or `enqueue_block_editor_assets` as outlined above. Editor-specific stylesheets should almost always be added with [`add_editor_style\(\)`](#) or [`wp_enqueue_block_style\(\)`](#).

The `wp_enqueue_block_style()` function allows you to load per-block stylesheets in the Editor and on the front end. Coupled with `theme.json`, this is one of the best methods of styling blocks. See the WordPress Developer Blog article [Leveraging theme.json and per-block styles for more performant themes](#) for more details.

[Backward compatibility and known issues](#)

As a general rule, when you enqueue assets in the iframed Editor, they will also be enqueued when the Editor is not iframed so long as you are using WordPress 6.3+. The opposite is not always true.

Suppose you are building a plugin or theme that requires backward compatibility to 6.2 or lower while maintaining compatibility with WordPress 6.3. In that case, you will not be able to use `enqueue_block_assets` since this hook does not enqueue assets in the content of the iframed Editor prior to 6.3.

As an alternative, you can use `enqueue_block_editor_assets` so long as the enqueued stylesheet contains at least one of the following selectors: `.editor-styles-wrapper`, `.wp-block`, or `.wp-block-*`. A warning message will be logged in the console, but the hook will apply the styles to the content of the Editor.

It's also important to note that as of WordPress 6.3, assets enqueued by `enqueue_block_assets` are loaded both inside and outside Editor's iframe for backward compatibility. Depending on the script libraries that you are trying to enqueue, this might cause problems. An ongoing discussion about this approach is happening in the Gutenberg [GitHub repository](#).

If you experience issues using any of the methods outlined in this guide that have not been previously reported, please [submit an issue](#) on GitHub.

First published

August 24, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Enqueueing assets in the Editor”](#)

[Previous Disable Editor functionality](#) [Previous: Disable Editor functionality](#)

[Next Feature Flags](#) [Next: Feature Flags](#)

Feature Flags

In this article

Table of Contents

- [Introducing process.env.IS_GUTENBERG_PLUGIN](#)
- [Basic usage](#)
 - [Exporting features](#)
 - [Importing features](#)
- [How it works](#)
 - [Dead code elimination](#)
- [Frequently asked questions](#)
 - [Why shouldn't I assign the result of an expression involving IS_GUTENBERG_PLUGIN to a variable, e.g. const isMyFeatureActive = process.env.IS_GUTENBERG_PLUGIN === 2?](#)

[↑ Back to top](#)

‘Feature flags’ are variables that allow you to prevent specific code in the Gutenberg project from being shipped to WordPress core, and to run certain experimental features only in the plugin.

Introducing process.env.IS_GUTENBERG_PLUGIN

The `process.env.IS_GUTENBERG_PLUGIN` is an environment variable whose value ‘flags’ whether code is running within the Gutenberg plugin.

When the codebase is built for the plugin, this variable will be set to `true`. When building for WordPress core, it will be set to `false` or `undefined`.

Basic usage

Exporting features

A plugin-only function or constant should be exported using the following ternary syntax:

```
function myPluginOnlyFeature() {  
    // implementation  
}  
  
export const pluginOnlyFeature =  
    process.env.IS_GUTENBERG_PLUGIN ? myPluginOnlyFeature : undefined;
```

In the above example, the `pluginOnlyFeature` export will be `undefined` in non-plugin environments such as WordPress core.

Importing features

If you’re attempting to import and call a plugin-only feature, be sure to wrap the function call in an `if` statement to avoid an error:

```
import { pluginOnlyFeature } from '@wordpress/foo';  
  
if ( process.env.IS_GUTENBERG_PLUGIN ) {  
    pluginOnlyFeature();  
}
```

How it works

During the webpack build, instances of `process.env.IS_GUTENBERG_PLUGIN` will be replaced using webpack’s [define plugin](#).

For example, in the following code –

```
if ( process.env.IS_GUTENBERG_PLUGIN ) {  
    pluginOnlyFeature();  
}
```

– the variable `process.env.IS_GUTENBERG_PLUGIN` will be replaced with the boolean `true` during the plugin-only build:

```
if ( true ) { // Wepack has replaced `process.env.IS_GUTENBERG_PLUGIN` with true  
    pluginOnlyFeature();  
}
```

This ensures that code within the body of the `if` statement will always be executed.

In WordPress core, the `process.env.IS_GUTENBERG_PLUGIN` variable is replaced with `undefined`. The built code looks like this:

```
if ( undefined ) { // Wepack has replaced `process.env.IS_GUTENBERG_PLUGIN`
    pluginOnlyFeature();
}
```

`undefined` evaluates to `false` so the plugin-only feature will not be executed.

Dead code elimination

For production builds, webpack '[minifies](#)' the code, removing as much unnecessary JavaScript as it can.

One of the steps involves something known as ‘dead code elimination’. For example, when the following code is encountered, webpack determines that the surrounding `if` statement is unnecessary:

```
if ( true ) {
    pluginOnlyFeature();
}
```

The condition will always evaluate to `true`, so webpack removes it, leaving behind the code that was in the body:

```
pluginOnlyFeature(); // The `if` condition block has been removed. Only the
```

Similarly, when building for WordPress core, the condition in the following `if` statement always resolves to false:

```
if ( undefined ) {
    pluginOnlyFeature();
}
```

In this case, the minification process will remove the entire `if` statement including the body, ensuring plugin-only code is not included in WordPress core build.

Frequently asked questions

Why shouldn't I assign the result of an expression involving `IS_GUTENBERG_PLUGIN` to a variable, e.g. `const isMyFeatureActive = process.env.IS_GUTENBERG_PLUGIN === 2?`?

Introducing complexity may prevent webpack’s minifier from identifying and therefore eliminating dead code. Therefore it is recommended to use the examples in this document to ensure your feature flag functions as intended. For further details, see the [Dead Code Elimination](#) section.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Feature Flags”](#)

[Previous](#) [Enqueueing assets in the Editor](#) [Previous: Enqueueing assets in the Editor](#)

[Next](#) [Formatting Toolbar API](#) [Next: Formatting Toolbar API](#)

Formatting Toolbar API

In this article

[Table of Contents](#)

- [Overview](#)
- [Before you start](#)
- [Step-by-step guide](#)
 - [Step 1: Register a new format](#)
 - [Step 2: Add a button to the toolbar](#)
 - [Step 3: Apply a format when clicked](#)
 - [Step 4: Show the button only for specific blocks \(Optional\)](#)
 - [Step 5: Add a button outside of the dropdown \(Optional\)](#)
- [Troubleshooting](#)
- [Additional resources](#)
- [Conclusion](#)

[↑ Back to top](#)

[Overview](#)

The Format API makes it possible for developers to add custom buttons to the formatting toolbar and have them apply a *format* to a text selection. Bold is an example of a standard button in the formatting toolbar.

Format API Test

You will rejoice to hear that no disaster has accompanied the enterprise which you have regarded with such yesterday, and my first task is to assure my dear s increasing confidence in the success of my undertakings.

Type / to choose a block

In WordPress lingo, a *format* is a [HTML tag with text-level semantics](#) used to give some special meaning to a text selection. For example, in this tutorial, the button to be hooked into the format toolbar will wrap a particular text selection with the `< samp>` HTML tag.

Before you start

This guide assumes you are already familiar with WordPress plugins and loading JavaScript with them, see the [Plugin Handbook](#) or [JavaScript Tutorial](#) to brush up.

You will need:

- WordPress development environment
- A minimal plugin activated and setup ready to edit
- JavaScript setup for building and enqueueing

The [complete format-api example](#) is available that you can use as a reference for your setup.

Step-by-step guide

The guide will refer to `src/index.js` as the JavaScript file where the changes are made. After each step, running `npm run build` creates `build/index.js` that is then loaded on the post editor screen.

Step 1: Register a new format

The first step is to register the new format, add `src/index.js` with the following:

```
import { registerFormatType } from '@wordpress/rich-text';

registerFormatType( 'my-custom-format/sample-output', {
    title: 'Sample output',
    tagName: 'samp',
    className: null,
} );
```

The list of available format types is maintained in the `core/rich-text` store. You can query the store to check that your custom format is now available.

Run this code in your browser's console to confirm:

```
wp.data.select( 'core/rich-text' ).getFormatTypes();
```

It'll return an array containing the format types, including your own.

Step 2: Add a button to the toolbar

With the format available, the next step is to add a button to the UI by registering a component for the `edit` property.

Using the `RichTextToolbarButton` component, update `src/index.js`:

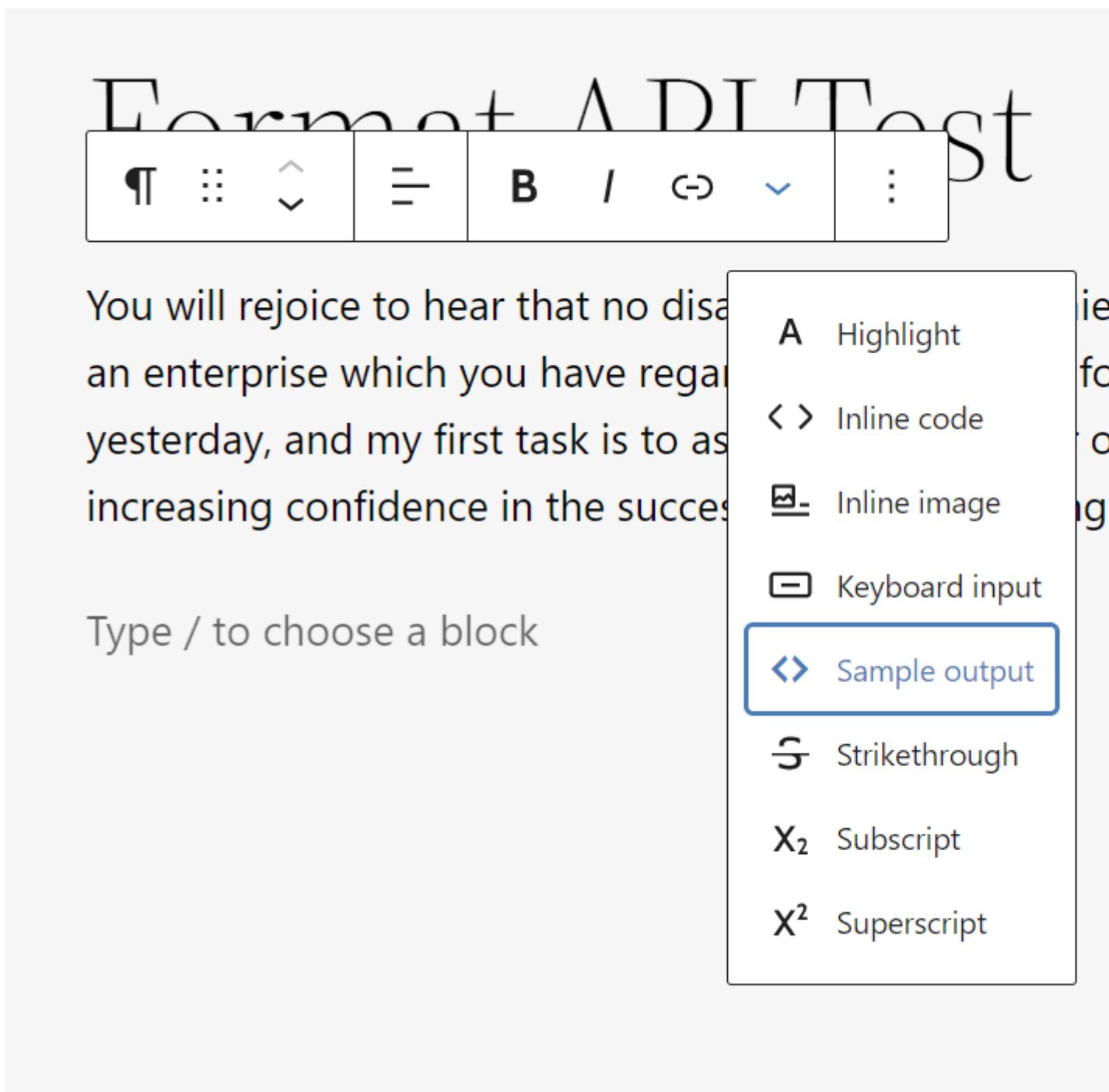
```
import { registerFormatType } from '@wordpress/rich-text';
import { RichTextToolbarButton } from '@wordpress/block-editor';

const MyCustomButton = ( props ) => {
    return (
        <RichTextToolbarButton
            icon="editor-code"
            title="Sample output"
            onClick={ () => {
                console.log( 'toggle format' );
            } }
        />
    );
};

registerFormatType( 'my-custom-format/sample-output', {
    title: 'Sample output',
}
```

```
    tagName: 'samp',
    className: null,
    edit: MyCustomButton,
} );
```

Let's check that everything is working as expected. Build and reload and then select any block containing text like for example the paragraph block. Confirm the new button was added to the format toolbar.



Click the button and check the console.log for the “toggle format” message.

If you do not see the button or message, double check you are building and loading the JavaScript properly; and check the console.log for any errors.

Step 3: Apply a format when clicked

Next is to update the button to apply a format when clicked.

For our example, the `<samp>` tag format is binary – either a text selection has the tag or not, so we can use the `toggleFormat` method from the `RichText` package.

Update `src/index.js` changing the `onClick` action:

```
import { registerFormatType, toggleFormat } from '@wordpress/rich-text';
import { RichTextToolbarButton } from '@wordpress/block-editor';

const MyCustomButton = ( { isActive, onChange, value } ) => {
    return (
        <RichTextToolbarButton
            icon="editor-code"
            title="Sample output"
            onClick={ () => {
                onChange(
                    toggleFormat( value, {
                        type: 'my-custom-format/sample-output',
                    } )
                );
            } }
            isActive={ isActive }
        />
    );
};

registerFormatType( 'my-custom-format/sample-output', {
    title: 'Sample output',
    tagName: 'samp',
    className: null,
    edit: MyCustomButton,
} );
```

Confirm it is working: first build and reload, then make a text selection and click the button. Your browser will likely display that selection differently than the surrounding text.

You can also confirm by switching to HTML view (Code editor `Ctrl+Shift+Alt+M`) and see the text selection wrapped with `<samp>` HTML tags.

Use the `className` option when registering to add your own custom class to the tag. You can use that class and custom CSS to target that element and style as you wish.

Step 4: Show the button only for specific blocks (Optional)

By default, the button is rendered on every rich text toolbar (image captions, buttons, paragraphs, etc). You can render the button only on blocks of a certain type by using [the data API](#).

Here is an example that only shows the button for Paragraph blocks:

```
import { registerFormatType, toggleFormat } from '@wordpress/rich-text';
import { RichTextToolbarButton } from '@wordpress/block-editor';
```

```

import { useSelect } from '@wordpress/data';

function ConditionalButton( { isActive, onChange, value } ) {
    const selectedBlock = useSelect( ( select ) => {
        return select( 'core/block-editor' ).getSelectedBlock();
    }, [] );

    if ( selectedBlock && selectedBlock.name !== 'core/paragraph' ) {
        return null;
    }

    return (
        <RichTextToolbarButton
            icon="editor-code"
            title="Sample output"
            onClick={ () => {
                onChange(
                    toggleFormat( value, {
                        type: 'my-custom-format/sample-output',
                    } )
                );
            } }
            isActive={ isActive }
        />
    );
}

registerFormatType( 'my-custom-format/sample-output', {
    title: 'Sample output',
    tagName: 'samp',
    className: null,
    edit: ConditionalButton,
} );

```

Step 5: Add a button outside of the dropdown (Optional)

Using the `RichTextToolbarButton` component, the button is added to the default dropdown menu. You can add the button directly to the toolbar by using the `BlockControls` component.

```

import { registerFormatType, toggleFormat } from '@wordpress/rich-text';
import { BlockControls } from '@wordpress/block-editor';
import { ToolbarGroup, ToolbarButton } from '@wordpress/components';

const MyCustomButton = ( { isActive, onChange, value } ) => {
    return (
        <BlockControls>
            <ToolbarGroup>
                <ToolbarButton
                    icon="editor-code"
                    title="Sample output"
                    onClick={ () => {
                        onChange(

```

```

        toggleFormat( value, {
          type: 'my-custom-format/sample-output',
        } )
      );
    } }
  isActive={ isActive }
/>
</ToolbarGroup>
</BlockControls>
);
};

registerFormatType( 'my-custom-format/sample-output', {
  title: 'Sample output',
  tagName: 'samp',
  className: null,
  edit: MyCustomButton,
} );

```

Troubleshooting

If you run into errors:

- Double check that you run `npm run build` first.
- Confirm no syntax errors or issues in build process.
- Confirm the JavaScript is loading in the editor.
- Check for any console error messages.

Additional resources

Reference documentation used in this guide:

- RichText: [registerFormatType](#)
- Components: [RichTextToolbarButton](#)
- RichText: [applyFormat](#)
- RichText: [removeFormat](#)
- RichText: [toggleFormat](#)

Conclusion

The guide showed you how to add a button to the toolbar and have it apply a format to the selected text. Try it out and see what you can build with it in your next plugin.

Download the [format-api example](#) from the [block-development-examples](#) repository.

First published

December 13, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Formatting Toolbar API”](#)

[Previous Feature Flags](#) [Previous: Feature Flags](#)

[Next Internationalization](#) [Next: Internationalization](#)

Internationalization

In this article

[Table of Contents](#)

- [What is internationalization?](#)
- [How to use i18n in JavaScript](#)
- [Provide your own translations](#)
 - [Create the translation file](#)
 - [Load the translation file](#)
 - [Test translations](#)
 - [Filtering translations](#)

[↑ Back to top](#)

What is internationalization?

Internationalization is the process to provide multiple language support to software, in this case WordPress. Internationalization is often abbreviated as **i18n**, where 18 stands for the number of letters between the first *i* and the last *n*.

Providing i18n support to your plugin and theme allows it to reach the largest possible audience, even without requiring you to provide the additional language translations. When you upload your software to WordPress.org, all JS and PHP files will automatically be parsed. Any detected translation strings are added to [translate.wordpress.org](#) to allow the community to translate, ensuring WordPress plugins and themes are available in as many languages as possible.

For PHP, WordPress has a long established process, see [How to Internationalize Your Plugin](#). The release of WordPress 5.0 brings a similar process for translation to JavaScript code.

How to use i18n in JavaScript

WordPress 5.0 introduced the wp-i18n JavaScript package that provides the functions needed to add translatable strings as you would in PHP.

First, add **wp-i18n** as a dependency when registering your script:

```
<?php
/**
 * Plugin Name: Myguten Plugin
 * Text Domain: myguten
 */
```

```

function myguten_block_init() {
    wp_register_script(
        'myguten-script',
        plugins_url( 'block.js', __FILE__ ),
        array( 'wp-blocks', 'react', 'wp-i18n', 'wp-block-editor' )
    );

    register_block_type( 'myguten/simple', array(
        'api_version' => 3,
        'editor_script' => 'myguten-script',
    ) );
}

add_action( 'init', 'myguten_block_init' );

```

In your code, you can include the i18n functions. The most common function is `__(double underscore)` which provides translation of a simple string. Here is a basic block example:

```

import { __ } from '@wordpress/i18n';
import { registerBlockType } from '@wordpress/blocks';
import { useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'myguten/simple', {
    apiVersion: 3,
    title: __( 'Simple Block', 'myguten' ),
    category: 'widgets',

    edit: () => {
        const blockProps = useBlockProps( { style: { color: 'red' } } );

        return <p { ...blockProps }>{ __( 'Hello World', 'myguten' ) }</p>
    },

    save: () => {
        const blockProps = useBlockProps.save( { style: { color: 'red' } } );

        return <p { ...blockProps }>{ __( 'Hello World', 'myguten' ) }</p>
    },
} );

```

In the above example, the function will use the first argument for the string to be translated. The second argument is the text domain which must match the text domain slug specified by your plugin.

Common functions available, these mirror their PHP counterparts are:

- `__('Hello World', 'my-text-domain')` – Translate a certain string.
- `_n('%s Comment', '%s Comments', numberOfComments, 'my-text-domain')` – Translate and retrieve the singular or plural form based on the supplied number.
- `_x('Default', 'block style', 'my-text-domain')` – Translate a certain string with some additional context.

Note: Every string displayed to the user should be wrapped in an i18n function.

After all strings in your code is wrapped, the final step is to tell WordPress your JavaScript contains translations, using the [wp_set_script_translations\(\)](#) function.

```
<?php
    function myguten_set_script_translations() {
        wp_set_script_translations( 'myguten-script', 'myguten' );
    }
    add_action( 'init', 'myguten_set_script_translations' );
```

This is all you need to make your plugin JavaScript code translatable.

When you set script translations for a handle WordPress will automatically figure out if a translations file exists on translate.wordpress.org, and if so ensure that it's loaded into `wp.i18n` before your script runs. With translate.wordpress.org, plugin authors also do not need to worry about setting up their own infrastructure for translations and can rely on a global community with dozens of active locales. Read more about [WordPress Translations](#).

Provide your own translations

You can create and ship your own translations with your plugin, if you have sufficient knowledge of the language(s) you can ensure the translations are available.

Create the translation file

The translation files must be in the JED 1.x JSON format.

To create a JED translation file, first you need to extract the strings from the text. Typically, the language files all live in a directory called `languages` in your plugin. Using [WP-CLI](#), you create a `.pot` file using the following command from within your plugin directory:

```
mkdir languages
wp i18n make-pot ./ languages/myguten.pot
```

This will create the file `myguten.pot` which contains all the translatable strings from your project.

```
msgid ""
msgstr ""
"Project-Id-Version: Scratch Plugin\n"
"Report-Msgid-Bugs-To: https://wordpress.org/support/plugin/scratch\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"POT-Creation-Date: 2019-03-08T11:26:56-08:00\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"X-Generator: WP-CLI 2.1.0\n"
"X-Domain: myguten\n"

#. Plugin Name of the plugin
msgid "Scratch Plugin"
msgstr ""
```

```

#: block.js:6
msgid "Simple Block"
msgstr ""

#: block.js:13
#: block.js:21
msgid "Hello World"
msgstr ""

```

Here, `msgid` is the string to be translated, and `msgstr` is the actual translation. In the POT file, `msgstr` will always be empty.

This POT file can then be used as the template for new translations. You should **copy the file** using the language code you are going to translate, this example will use the Esperanto (eo) language:

```
cp myguten.pot myguten-eo.po
```

For this simple example, you can simply edit the `.po` file in your editor and add the translation to all the `msgstr` sets. For a larger, more complex set of translation, the [GlotPress](#) and [Poedit](#) tools exist to help.

You need also to add the `Language: eo` parameter. Here is full `myguten-eo.po` translated file

```

# Copyright (C) 2019
# This file is distributed under the same license as the Scratch Plugin pl
msgid ""
msgstr ""

"Project-Id-Version: Scratch Plugin\n"
"Report-Msgid-Bugs-To: https://wordpress.org/support/plugin/scratch\n"
"Last-Translator: Marcus Kazmierczak <marcus@mkaz.com>\n"
"Language-Team: Esperanto <marcus@mkaz.com>\n"
"Language: eo\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"POT-Creation-Date: 2019-02-18T07:20:46-08:00\n"
"PO-Revision-Date: 2019-02-18 08:16-0800\n"
"X-Generator: Poedit 2.2.1\n"
"X-Domain: myguten\n"

#. Plugin Name of the plugin
msgid "Scratch Plugin"
msgstr "Scratch kromprogrameto"

#: block.js:6
msgid "Simple Block"
msgstr "Simpla bloko"

#: block.js:13 block.js:21
msgid "Hello World"
msgstr "Saluton mundo"

```

The last step to create the translation file is to convert the `myguten-eo.po` to the JSON format needed. For this, you can use WP-CLI's [wp_i18n_make-json command](#), which requires WP-CLI v2.2.0 and later.

```
wp i18n make-json myguten-eo.po --no-purge
```

This will generate the JSON file `myguten-eo-[md5].json` with the contents:

```
{
    "translation-revision-date": "2019-04-26T13:30:11-07:00",
    "generator": "WP-CLI/2.2.0",
    "source": "block.js",
    "domain": "messages",
    "locale_data": {
        "messages": {
            "": {
                "domain": "messages",
                "lang": "eo",
                "plural-forms": "nplurals=2; plural=(n != 1);"
            },
            "Simple Block": [ "Simpla Bloko" ],
            "Hello World": [ "Salunton mondo" ]
        }
    }
}
```

[Load the translation file](#)

The final part is to tell WordPress where it can look to find the translation file. The `wp_set_script_translations` function accepts an optional third argument that is the path it will first check for translations. For example:

```
<?php
    function myguten_set_script_translations() {
        wp_set_script_translations( 'myguten-script', 'myguten', plugin_dir_path( __FILE__ ) );
    }
    add_action( 'init', 'myguten_set_script_translations' );
```

WordPress will check for a file in that path with the format `${domain}- ${locale}- ${handle}.json` as the source of translations. Alternatively, instead of the registered handle you can use the md5 hash of the relative path of the file, `${domain}- ${locale}` in the form of `${domain}- ${locale}- ${md5}.json`.

Using `make-json` automatically names the file with the md5 hash, so it is ready as-is. You could rename the file to use the handle instead, in which case the file name would be `myguten-eo-myguten-script.json`.

[Test translations](#)

You will need to set your WordPress installation to Esperanto language. Go to Settings > General and change your site language to Esperanto.

With the language set, create a new post, add the block, and you will see the translations used.

[Filtering translations](#)

The outputs of the translation functions (`__()`, `_x()`, `_n()`, and `_nx()`) are filterable, see [i18n Filters](#) for full information.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Internationalization”](#)

[Previous Formatting Toolbar API](#) [Previous: Formatting Toolbar API](#)

[Next Meta Boxes](#) [Next: Meta Boxes](#)

Meta Boxes

In this article

Table of Contents

- [Overview](#)
 - [Use blocks to store meta](#)
- [Before you start](#)
- [Step-by-step guide](#)
 - [Step 1: Register meta field](#)
 - [Step 2: Add meta block](#)
 - [Step 3: Use post meta data](#)
 - [Step 4: Use block templates \(optional\)](#)
- [Conclusion](#)
- [Backward compatibility](#)
 - [Testing, converting, and maintaining existing meta boxes](#)
 - [Meta box data collection](#)
 - [Redux and React meta box management](#)
 - [Common compatibility issues](#)

[↑ Back to top](#)

[Overview](#)

Prior to the block editor, custom meta boxes were used to extend the editor. Now there are new ways to extend, giving more power to the developer and a better experience for the authors. It is recommended to port older custom meta boxes to one of these new methods to create a more unified and consistent experience for those using the editor.

The block editor does support most existing meta boxes, see [the backward compatibility section below](#) for details .

If you are interested in working with the post meta outside the editor, check out the [Sidebar Tutorial](#).

[Use blocks to store meta](#)

Typically, blocks store attribute values in the serialized block HTML. However, you can also create a block that saves its attribute values as post meta, that can be accessed programmatically anywhere in your template.

This guide shows how to create a block that prompts a user for a single value, and saves it to post meta.

[Before you start](#)

This guide assumes you are already familiar with WordPress plugins, post meta, and basic JavaScript. Review the [Getting started with JavaScript tutorial](#) for an introduction.

The guide will walk through creating a basic block, but recommended to go through the [Create Block tutorial](#) for a deeper understanding of creating custom blocks.

You will need:

- WordPress development environment,
- A minimal plugin activated and ready to edit
- JavaScript setup for building and enqueueing

A [complete meta-block example](#) is available that you can use as a reference for your setup.

[Step-by-step guide](#)

1. [Register Meta Field](#)
2. [Add Meta Block](#)
3. [Use Post Meta Data](#)
4. [Finishing Touches](#)

[Step 1: Register meta field](#)

A post meta field is a WordPress object used to store extra data about a post. You need to first register a new meta field prior to use. See Managing [Post Metadata](#) to learn more about post meta.

When registering the field, note the `show_in_rest` parameter. This ensures the data will be included in the REST API, which the block editor uses to load and save meta data. See the [register_post_meta](#) function definition for extra information.

Additionally, your post type needs to support `custom-fields` for `register_post_meta` function to work

To register the field, add the following to your PHP plugin:

```

<?php
// register custom meta tag field
function myguten_register_post_meta() {
    register_post_meta( 'post', 'myguten_meta_block_field', array(
        'show_in_rest' => true,
        'single' => true,
        'type' => 'string',
    ) );
}
add_action( 'init', 'myguten_register_post_meta' );

```

Step 2: Add meta block

With the meta field registered in the previous step, next create a new block to display the field value to the user.

The hook `useEntityProp` can be used by the blocks to get or change meta values.

Add this code to the JavaScript `src/index.js`:

```

import { registerBlockType } from '@wordpress/blocks';
import { TextControl } from '@wordpress/components';
import { useSelect } from '@wordpress/data';
import { useEntityProp } from '@wordpress/core-data';
import { useBlockProps } from '@wordpress/block-editor';

registerBlockType( 'myguten/meta-block', {
    edit: ( { setAttributes, attributes } ) => {
        const blockProps = useBlockProps();
        const postType = useSelect(
            ( select ) => select( 'core/editor' ).getCurrentPostType(),
            []
        );

        const [ meta, setMeta ] = useEntityProp( 'postType', postType, 'me'

        const metaFieldValue = meta[ 'myguten_meta_block_field' ];
        const updateMetaValue = ( newValue ) => {
            setMeta( { ...meta, myguten_meta_block_field: newValue } );
        };

        return (
            <div { ...blockProps }>
                <TextControl
                    label="Meta Block Field"
                    value={ metaFieldValue }
                    onChange={ updateMetaValue }
                />
            </div>
        );
    },
    // No information saved to the block.
    // Data is saved to post meta via the hook.

```

```
        save: () => {
          return null;
        },
      } );
    }
  );
```

Confirm this works by creating a post and add the Meta Block. You will see your field that you can type a value in. When you save the post, either as a draft or published, the post meta value will be saved too. You can verify by saving and reloading your draft, the form will still be filled in on reload.

You could also confirm the data is saved by checking the database table `wp_postmeta` and confirm the new post id contains the new field data.

Troubleshooting: Be sure to build your code between changes, you updated the PHP code from Step 1, and JavaScript files are enqueued. Check the build output and developer console for errors.

[Step 3: Use post meta data](#)

You can use the post meta data stored in the last step in multiple ways.

Use post meta in PHP

The first example uses the value from the post meta field and appends it to the end of the post content wrapped in a H4 tag.

```
function myguten_content_filter( $content ) {
  $value = get_post_meta( get_the_ID(), 'myguten_meta_block_field', true );
  if ( $value ) {
    return sprintf( "%s <h4> %s </h4>", $content, esc_html( $value ) );
  } else {
    return $content;
  }
}
add_filter( 'the_content', 'myguten_content_filter' );
```

Use post meta in a block

You can also use the post meta data in other blocks. For this example the data is loaded at the end of every Paragraph block when it is rendered, ie. shown to the user. You can replace this for any core or custom block types as needed.

In PHP, use the [register_block_type](#) function to set a callback when the block is rendered to include the meta value.

```
function myguten_render_paragraph( $block_attributes, $content ) {
  $value = get_post_meta( get_the_ID(), 'myguten_meta_block_field', true );
  // check value is set before outputting
  if ( $value ) {
    return sprintf( "%s (%s)", $content, esc_html( $value ) );
  } else {
    return $content;
  }
}
```

```
register_block_type( 'core/paragraph', array(
    'api_version' => 3,
    'render_callback' => 'myguten_render_paragraph',
) );
```

Step 4: Use block templates (optional)

One problem using a meta block is the block is easy for an author to forget, since it requires being added to each post. You solve this by using [block templates](#). A block template is a predefined list of block items per post type. Templates allow you to specify a default initial state for a post type.

For this example, you use a template to automatically insert the meta block at the top of a post.

Add the following code to the `myguten-meta-block.php` file:

```
function myguten_register_template() {
    $post_type_object = get_post_type_object( 'post' );
    $post_type_object->template = array(
        array( 'myguten/meta-block' ),
    );
}
add_action( 'init', 'myguten_register_template' );
```

You can also add other block types in the array, including placeholders, or even lock down a post to a set of specific blocks. Templates are a powerful tool for controlling the editing experience, see the documentation linked above for more.

Conclusion

This guide showed how using blocks you can read and write to post meta. See the section below for backward compatibility with existing meta boxes.

Backward compatibility

Testing, converting, and maintaining existing meta boxes

Before converting meta boxes to blocks, it may be easier to test if a meta box works with the block editor, and explicitly mark it as such.

If a meta box *doesn't* work with the block editor, and updating it to work correctly is not an option, the next step is to add the `__block_editor_compatible_meta_box` argument to the meta box declaration:

```
add_meta_box( 'my-meta-box', 'My Meta Box', 'my_meta_box_callback',
    null, 'normal', 'high',
    array(
        '__block_editor_compatible_meta_box' => false,
    )
);
```

WordPress won't show the meta box but a message saying that it isn't compatible with the block editor, including a link to the Classic Editor plugin. By default, `__block_editor_compatible_meta_box` is `true`.

After a meta box is converted to a block, it can be declared as existing for backward compatibility:

```
add_meta_box( 'my-meta-box', 'My Meta Box', 'my_meta_box_callback',
    null, 'normal', 'high',
    array(
        '__back_compat_meta_box' => true,
    )
);
```

When the block editor is used, this meta box will no longer be displayed in the meta box area, as it now only exists for backward compatibility purposes. It will display as before in the classic editor.

[Meta box data collection](#)

On each block editor page load, we register an action that collects the meta box data to determine if an area is empty. The original global state is reset upon collection of meta box data.

See [`register_and_do_post_meta_boxes`](#).

It will run through the functions and hooks that `post.php` runs to register meta boxes; namely `add_meta_boxes`, `add_meta_boxes_{$post->post_type}`, and `do_meta_boxes`.

Meta boxes are filtered to strip out any core meta boxes, standard custom taxonomy meta boxes, and any meta boxes that have declared themselves as only existing for backward compatibility purposes.

Then each location for this particular type of meta box is checked for whether it is active. If it is not empty a value of `true` is stored, if it is empty a value of `false` is stored. This meta box location data is then dispatched by the editor Redux store in `INITIALIZE_META_BOX_STATE`.

Ideally, this could be done at instantiation of the editor and help simplify this flow. However, it is not possible to know the meta box state before `admin_enqueue_scripts`, where we are calling `initializeEditor()`. This will have to do, unless we want to move `initializeEditor()` to fire in the footer or at some point after `admin_head`. With recent changes to editor bootstrapping this might now be possible. Test with ACF to make sure.

[Redux and React meta box management](#)

When rendering the block editor, the meta boxes are rendered to a hidden div `#metaboxes`.

The Redux store will hold all meta boxes as inactive by default. When `INITIALIZE_META_BOX_STATE` comes in, the store will update any active meta box areas by setting the `isActive` flag to `true`. Once this happens React will check for the new props sent in by Redux on the `MetaBox` component. If that `MetaBox` is now active, instead of rendering `null`, a `MetaBoxArea` component will be rendered. The `MetaBox` component is the container component that mediates between the `MetaBoxArea` and the Redux Store. If no meta boxes are active, nothing happens. This will be the default behavior, as all core meta boxes have been stripped.

MetaBoxArea component

When the component renders it will store a reference to the meta boxes container and retrieve the meta boxes HTML from the prefetch location.

When the post is updated, only meta box areas that are active will be submitted. This prevents unnecessary requests. No extra revisions are created by the meta box submissions. A Redux action will trigger on REQUEST_POST_UPDATE for any active meta box. See `editor/effects.js`. The REQUEST_META_BOX_UPDATES action will set that meta box's state to `isUpdating`. The `isUpdating` prop will be sent into the `MetaBoxArea` and cause a form submission.

When the meta box area is saving, we display an updating overlay, to prevent users from changing the form values while a save is in progress.

An example save url would look like:

```
mysite.com/wp-admin/post.php?post=1&action=edit&meta-box-loader=1
```

This url is automatically passed into React via a `_wpMetaBoxUrl` global variable.

This page mimics the `post.php` post form, so when it is submitted it will fire all of the normal hooks and actions, and have the proper global state to correctly fire any PHP meta box mumbo jumbo without needing to modify any existing code. On successful submission, React will signal a `handleMetaBoxReload` to remove the updating overlay.

Common compatibility issues

Most PHP meta boxes should continue to work in the block editor, but some meta boxes that include advanced functionality could break. Here are some common reasons why meta boxes might not work as expected in the block editor:

- Plugins relying on selectors that target the post title, post content fields, and other metaboxes (of the old editor).
- Plugins relying on TinyMCE's API because there's no longer a single TinyMCE instance to talk to in the block editor.
- Plugins making updates to their DOM on “submit” or on “save”.

Please also note that if your plugin triggers a PHP warning or notice to be output on the page, this will cause the HTML document type (`<!DOCTYPE html>`) to be output incorrectly. This will cause the browser to render using “Quirks Mode”, which is a compatibility layer that gets enabled when the browser doesn't know what type of document it is parsing. The block editor is not meant to work in this mode, but it can *appear* to be working just fine. If you encounter issues such as *meta boxes overlaying the editor* or other layout issues, please check the raw page source of your document to see that the document type definition is the first thing output on the page. There will also be a warning in the JavaScript console, noting the issue.

First published

December 27, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Meta Boxes”](#)

[Previous Internationalization](#) [Previous: Internationalization](#)

[Next Notices](#) [Next: Notices](#)

Notices

In this article

[Table of Contents](#)

- [Notices in the Classic Editor](#)
- [Notices in the Block Editor](#)
- [Learn more](#)

[↑ Back to top](#)

Notices are informational UI displayed near the top of admin pages. WordPress core, themes, and plugins all use notices to indicate the result of an action, or to draw the user’s attention to necessary information.

In the classic editor, notices hooked onto the `admin_notices` action can render whatever HTML they’d like. In the block editor, notices are restricted to a more formal API.

[**Notices in the Classic Editor**](#)

In the classic editor, here’s an example of the “Post draft updated” notice:

Add New Post

Post draft updated. [Preview post](#)

Enter title here



Add Media

Paragraph ▾

B

I

≡

1
2
3

“

≡

Producing an equivalent “Post draft updated” notice would require code like this:

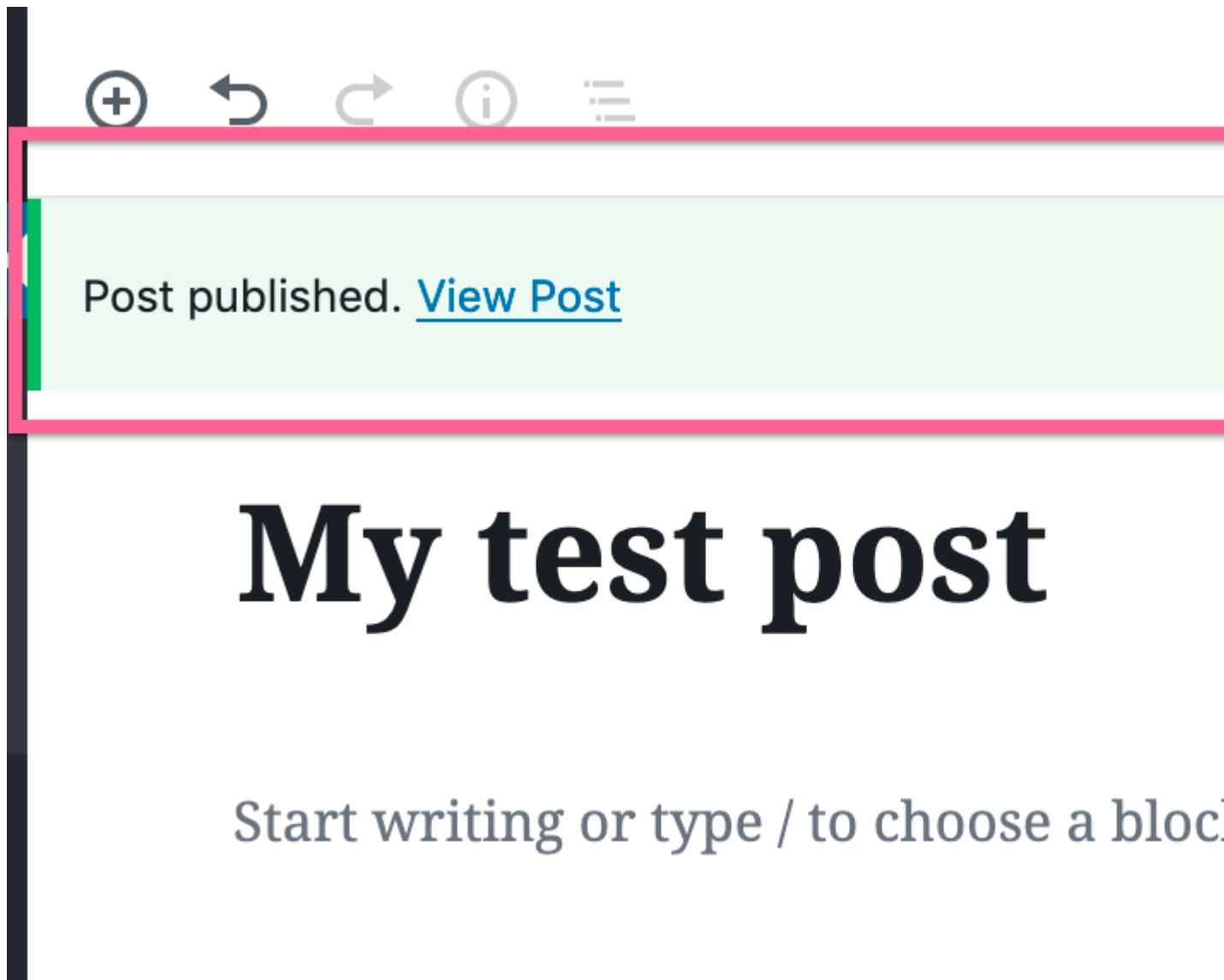
```
/**  
 * Hook into the 'admin_notices' action to render  
 * a generic HTML notice.  
 */  
function myguten_admin_notice() {  
    $screen = get_current_screen();  
    // Only render this notice in the post editor.  
    if ( ! $screen || 'post' !== $screen->base ) {  
        return;  
    }  
    // Render the notice's HTML.  
    // Each notice should be wrapped in a <div>  
    // with a 'notice' class.  
    echo '<div class="notice notice-success is-dismissible"><p>';
```

```
echo sprintf( __( 'Post draft updated. <a href="%s" target="_blank">Pr  
echo '</p></div>' );  
};  
add_action( 'admin_notices', 'myguten_admin_notice' );
```

Importantly, the `admin_notices` hook allows a developer to render whatever HTML they'd like. One advantage is that the developer has a great amount of flexibility. The key disadvantage is that arbitrary HTML makes future iterations on notices more difficult, if not possible, because the iterations need to accommodate for arbitrary HTML. This is why the block editor has a formal API.

Notices in the Block Editor

In the block editor, here's an example of the "Post published" notice:



Producing an equivalent "Post published" notice would require code like this:

```
( function ( wp ) {  
    wp.data.dispatch( 'core/notices' ).createNotice(  
        'success', // Can be one of: success, info, warning, error.
```

```
'Post published.', // Text string to display.  
}  
    isDismissible: true, // Whether the user can dismiss the notice  
    // Any actions the user can perform.  
    actions: [  
        {  
            url: '#',  
            label: 'View post',  
        },  
    ],  
},  
);  
} )( window.wp );
```

You'll want to use this *Notices Data API* when producing a notice from within the JavaScript application lifecycle.

To better understand the specific code example above:

- wp is WordPress global window variable.
- wp.data is an object provided by the block editor for accessing the block editor data store.
- wp.data.dispatch('core/notices') accesses functionality registered to the block editor data store by the Notices package.
- createNotice() is a function offered by the Notices package to register a new notice. The block editor reads from the notice data store in order to know which notices to display.

Check out the [Loading JavaScript](#) tutorial for a primer on how to load your custom JavaScript into the block editor.

[Learn more](#)

The block editor offers a complete API for generating notices. The official documentation is a great place to review what's possible.

For a full list of the available actions and selectors, refer to the [Notices Data Handbook](#) page.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Notices”](#)

[Previous Meta Boxes](#) [Previous: Meta Boxes](#)

[Next Plugin Sidebar](#) [Next: Plugin Sidebar](#)

Plugin Sidebar

In this article

Table of Contents

- [Overview](#)
- [Before you start](#)
- [Step-by-step guide](#)
 - [Step 1: Get a sidebar up and running](#)
 - [Step 2: Tweak the sidebar style and add controls](#)
 - [Step 3: Register the meta field](#)
 - [Step 4: Initialize the input control](#)
 - [Step 5: Update the meta field when the input's content changes](#)
- [Additional resources](#)
- [Conclusion](#)
 - [Note](#)

[↑ Back to top](#)

Overview

How to add a sidebar to your plugin. A sidebar is the region to the far right of the editor. Your plugin can add an additional icon next to the InspectorControls (gear icon) that can be expanded.



Gutenberg Dev

10

0

New

View Post



✓ Saved



Welcome to the Gutenberg Editor



The goal of this new editor is to make adding rich content to WordPress simple and enjoyable. This whole post is composed of *pieces of content*

Note: this tutorial covers a custom sidebar; if you are looking to add controls to the sidebar see the [Block Toolbar and Settings Sidebar](#)

Before you start

The tutorial assumes you have an existing plugin setup and are ready to add PHP and JavaScript code. Please, refer to [Getting started with JavaScript](#) tutorial for an introduction to WordPress plugins and how to use JavaScript to extend the block editor.

Step-by-step guide

Step 1: Get a sidebar up and running

The first step is to tell the editor that there is a new plugin that will have its own sidebar. Use the [registerPlugin](#), [PluginSidebar](#), and [createElement](#) utilities provided by the `@wordpress/plugins`, `@wordpress/edit-post`, and `react` packages, respectively.

Add the following code to a JavaScript file called `plugin-sidebar.js` and save it within your plugin's directory:

```
( function( wp, React ) {
    var el = React.createElement;
    var registerPlugin = wp.plugins.registerPlugin;
    var PluginSidebar = wp.editPost.PluginSidebar;

    registerPlugin( 'my-plugin-sidebar', {
        render: function() {
            return el(
                PluginSidebar,
                {
                    name: 'my-plugin-sidebar',
                    icon: 'admin-post',
                    title: 'My plugin sidebar',
                },
                'Meta field'
            );
        },
    } );
} )( window.wp, window.React );
```

For this code to work, those utilities need to be available in the browser, so you must specify `wp-plugins`, `wp-edit-post`, and `react` as dependencies of your script.

Here is the PHP code to register your script and specify the dependencies:

```
<?php

/*
Plugin Name: Sidebar plugin
*/

function sidebar_plugin_register() {
    wp_register_script(
        'plugin-sidebar-js',
        plugins_url( 'plugin-sidebar.js', __FILE__ ),
        array( 'wp-plugins', 'wp-edit-post', 'react' )
    );
}
```

```
        );
    }
add_action( 'init', 'sidebar_plugin_register' );

function sidebar_plugin_script_enqueue() {
    wp_enqueue_script( 'plugin-sidebar-js' );
}
add_action( 'enqueue_block_editor_assets', 'sidebar_plugin_script_enqueue'
```

After installing and activating this plugin, there is a new icon resembling a tack in the top-right of the editor. Upon clicking it, the plugin's sidebar will be opened:



Gutenberg Dev

10

0

New

View Post



✓ Saved



Welcome to the Gutenberg Editor



Of Mountains & Printing Presses

The goal of this new editor is to make adding rich content to WordPress simple and enjoyable. This whole post is composed of *pieces of content*

Step 2: Tweak the sidebar style and add controls

After the sidebar is up and running, the next step is to fill it up with the necessary components and basic styling.

To visualize and edit the meta field value you'll use an input component. The `@wordpress/components` package contains many components available for you to reuse, and, specifically, the [TextControl](#) is aimed at creating an input field:

```
( function ( wp ) {
    var el = React.createElement;
    var registerPlugin = wp.plugins.registerPlugin;
    var PluginSidebar = wp.editPost.PluginSidebar;
    var TextControl = wp.components.TextControl;

    registerPlugin( 'my-plugin-sidebar' , {
        render: function () {
            return el(
                PluginSidebar,
                {
                    name: 'my-plugin-sidebar',
                    icon: 'admin-post',
                    title: 'My plugin sidebar',
                },
                el(
                    'div',
                    { className: 'plugin-sidebar-content' },
                    el( TextControl, {
                        label: 'Meta Block Field',
                        value: 'Initial value',
                        onChange: function ( content ) {
                            console.log( 'content changed to ' , content );
                        },
                    } )
                )
            );
        },
    } );
} )( window.wp );
```

Update the `plugin-sidebar.js` with this new code. Notice that it uses a new utility called `wp.components` from the `@wordpress/components` package. Be sure to add `wp-components` to the dependencies in the `wp_register_script` function in the PHP file.

The code introduces:

- a CSS class `plugin-sidebar-content` to the `div` element to target styles,
- a `TextControl` component instead of the plain 'Meta field' text.

With the new CSS class available you can add a little style. Create a new file in your plugin directory called `plugin-sidebar.css` with the following to give some padding:

```
.plugin-sidebar-content {
    padding: 16px;
}
```

Register the script and enqueue it to load with `enqueue_block_editor_assets` alongside the JavaScript file.

After those changes, the PHP code will look like this:

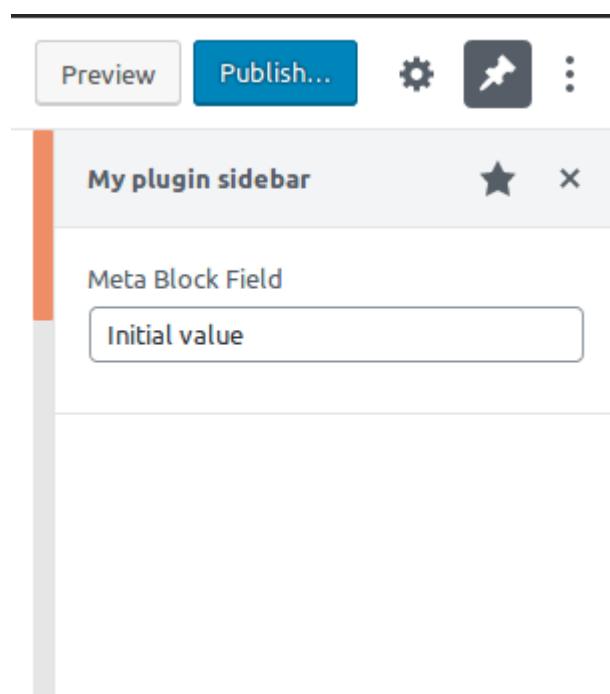
```
<?php

/*
Plugin Name: Sidebar example
*/

function sidebar_plugin_register() {
    wp_register_script(
        'plugin-sidebar-js',
        plugins_url( 'plugin-sidebar.js', __FILE__ ),
        array(
            'react',
            'wp-plugins',
            'wp-edit-post',
            'wp-components'
        )
    );
    wp_register_style(
        'plugin-sidebar-css',
        plugins_url( 'plugin-sidebar.css', __FILE__ )
    );
}
add_action( 'init', 'sidebar_plugin_register' );

function sidebar_plugin_script_enqueue() {
    wp_enqueue_script( 'plugin-sidebar-js' );
    wp_enqueue_style( 'plugin-sidebar-css' );
}
add_action( 'enqueue_block_editor_assets', 'sidebar_plugin_script_enqueue'
```

Reload the editor and open the sidebar:



This code doesn't let users store or retrieve data just yet, so the next steps will focus on how to connect it to the meta block field.

Step 3: Register the meta field

To work with fields in the `post_meta` table, use the [register_post_meta](#) function to create a new field called `sidebar_plugin_meta_block_field`.

Note: this field needs to be available to the REST API because that's how the block editor access data.

Add the PHP code in your plugins `init` callback function:

```
register_post_meta( 'post', 'sidebar_plugin_meta_block_field', array(
    'show_in_rest' => true,
    'single' => true,
    'type' => 'string',
) );
```

To confirm, query the block editor store to see the field is loaded. After implementing, reload the editor page and open your browser's developer console. Use this JavaScript snippet in the console to confirm:

```
wp.data.select( 'core/editor' ).getCurrentPost().meta;
```

The function will return an object containing the registered meta field you registered.

If the code returns `undefined` make sure your post type supports `custom-fields`. Either when [registering the post](#) or with [add_post_type_support function](#).

Step 4: Initialize the input control

With the field available in the editor store, it can now be surfaced to the UI. We extract the input control to a function to keep the code clean as we add functionality.

```
( function ( wp ) {
    var el = React.createElement;
    var registerPlugin = wp.plugins.registerPlugin;
    var PluginSidebar = wp.editPost.PluginSidebar;
    var TextControl = wp.components.TextControl;

    var MetaBlockField = function () {
        return el( TextControl, {
            label: 'Meta Block Field',
            value: 'Initial value',
            onChange: function ( content ) {
                console.log( 'content changed to ', content );
            },
        } );
    };

    registerPlugin( 'my-plugin-sidebar', {
        render: function () {
            return el(
```

```

        PluginSidebar,
    {
        name: 'my-plugin-sidebar',
        icon: 'admin-post',
        title: 'My plugin sidebar',
    },
    el(
        'div',
        { className: 'plugin-sidebar-content' },
        el( MetaBlockField )
    )
);
},
) );
} )( window.wp );

```

We want to initialize the value in the `MetaBlockField` component with the value of `sidebar_plugin_meta_block_field`, and keep it updated when that value changes.

The `useSelect` function is used to fetch data when the component loads and will update if the data changes. Here is the code update with `useSelect`:

```

( function ( wp ) {
    var el = React.createElement;
    var registerPlugin = wp.plugins.registerPlugin;
    var PluginSidebar = wp.editPost.PluginSidebar;
    var Text = wp.components.TextControl;
    var useSelect = wp.data.useSelect;

    var MetaBlockField = function () {
        var metaFieldValue = useSelect( function ( select ) {
            return select( 'core/editor' ).getEditedPostAttribute(
                'meta'
            )[ 'sidebar_plugin_meta_block_field' ];
        }, [] );

        return el( Text, {
            label: 'Meta Block Field',
            value: metaFieldValue,
            onChange: function ( content ) {
                console.log( 'content has changed to ', content );
            },
        } );
    };

    registerPlugin( 'my-plugin-sidebar', {
        render: function () {
            return el(
                PluginSidebar,
                {
                    name: 'my-plugin-sidebar',
                    icon: 'admin-post',
                    title: 'My plugin sidebar',
                },
            );
        }
    });
}
);

```

```

        el(
            'div',
            { className: 'plugin-sidebar-content' },
            el( MetaBlockField )
        )
    );
},
} );
} )( window.wp );

```

The `wp.data.useSelect` function is from the `@wordpress/data` package, so `wp-data` needs to be added as a dependency in the `wp_register_script` function in PHP.

Note: The `getEditedPostAttribute` call is used to retrieve the most recent values of the post, including user editions that haven't been yet saved.

Confirm it's working by updating the code, reloading, and opening the sidebar. The input's content is no longer `Initial value` but a void string. Users can't type values yet, but you can check that the component is updated if the value in the store changes. Open the browser's console, execute

```

wp.data
    .dispatch( 'core/editor' )
    .editPost( { meta: { sidebar_plugin_meta_block_field: 'hello world!' } }

```

You can observe the content changing in the input component.

Step 5: Update the meta field when the input's content changes

The last step is to update the meta field when the input content changes.

The `useDispatch` function takes a store name as its only argument and returns methods that you can use to update the store, in this case we'll use `editPost`

```

( function ( wp ) {
    var el = React.createElement;
    var registerPlugin = wp.plugins.registerPlugin;
    var PluginSidebar = wp.editPost.PluginSidebar;
    var TextControl = wp.components.TextControl;
    var useSelect = wp.data.useSelect;
    var useDispatch = wp.data.useDispatch;

    var MetaBlockField = function ( props ) {
        var metaFieldValue = useSelect( function ( select ) {
            return select( 'core/editor' ).getEditedPostAttribute(
                'meta'
            )[ 'sidebar_plugin_meta_block_field' ];
        }, [] );
        var editPost = useDispatch( 'core/editor' ).editPost;

        return el( TextControl, {
            label: 'Meta Block Field',
            value: metaFieldValue,
            onChange: function ( content ) {

```

```

        editPost( {
            meta: { sidebar_plugin_meta_block_field: content },
        } );
    },
} );
};

registerPlugin( 'my-plugin-sidebar', {
    render: function () {
        return el(
            PluginSidebar,
            {
                name: 'my-plugin-sidebar',
                icon: 'admin-post',
                title: 'My plugin sidebar',
            },
            el(
                'div',
                { className: 'plugin-sidebar-content' },
                el( MetaBlockField )
            )
        );
    },
} );
} )( window.wp );

```

After the update, when the user types, the input control calls `editPost` and updates the editor store on each keystroke.

Update the JavaScript, load the sidebar, and type in the input field. You can confirm it is saved by typing something in the input control and executing the JavaScript snippet in your browser's development console:

```
wp.data.select( 'core/editor' ).getEditedPostAttribute( 'meta' )[
    'sidebar_plugin_meta_block_field'
];
```

The message displayed should be what you typed in the input.

When saving a post, you can confirm it is stored properly in the database by reloading after a save and confirming the input control is initialized with the last value you typed.

Additional resources

Documentation for working with the [@wordpress/data package](#).

Functions used in this guide:

- [useSelect](#).
- [getEditedPostAttribute](#)
- [useDispatch](#)

Conclusion

You now have a custom sidebar that you can use to update `post_meta` content.

A complete example is available, download the [plugin-sidebar example](#) from the [block-development-examples](#) repository.

Note

If you have enabled Custom Fields in the ‘Panels’ page of the Editor ‘Preferences’ (via the three dots in top right), a field with the same name as the TextControl, in this case `sidebar_plugin_meta_block_field`, will also appear in the custom fields panel at the bottom of the editor window. These two fields have access to the same meta property.

Add title

Plugin Sidebar – hello from the editor!!

Custom Fields

Name
sidebar_plugin_meta_block_field

[Delete](#) [Update](#)

hello

On saving the post the value in the TextControl will be saved first and the value in the custom field will be saved second, so that is the one that ends up persisting in the database. So if you change the value in the TextControl it is still the one in the custom field that ends up getting saved.

This problem does not exist if Custom Fields is not enabled.

If you need to have Custom Fields enabled and also have post meta in the sidebar there are two possible solutions:

1. Precede the name of the meta field with an underscore, so the name in the above example would be `_sidebar_plugin_meta_block_field`. This indicates that the post meta should be treated as private so it will not be visible in the Custom Fields section of a post. With this solution an error will be generated when you save the post unless you add an `auth_callback` property to the `args` array passed to `register_post_meta` with a function that ultimately returns `true`. See the `args` documentation in the [post_meta](#) page for more info.
2. In the TextControl's `onChange` function, target the Value field textarea and set the value there to be the same as the value in the TextControl meta field. The value will then be identical in both places and so you can be assured that if the value is changed in the TextControl then it will still be saved to the database.

```
return el( TextControl, {
  label: 'Meta Block Field',
  value: metaFieldValue,
  onChange: function ( content ) {
    editPost( {
      meta: { sidebar_plugin_meta_block_field: content }
    })
    document.querySelector( {the-value-textarea} ).innerHTML = content;
  },
});
```

First published

December 20, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Plugin Sidebar](#)

[Previous Notices](#) [Previous: Notices](#)

[Next Propagating updates for block types](#) [Next: Propagating updates for block types](#)

Propagating updates for block types

In this article

Table of Contents

- [Recommendations on managing updates](#)
 - [Establish early what content you expect to require updates](#)
 - [Embrace theme design at the block level](#)

- [Content types \(and how to properly update them\)](#)
 - [Blocks](#)
 - [Patterns](#)
 - [Reusable blocks](#)
 - [Template parts and templates](#)
 - [Resources](#)

[↑ Back to top](#)

This resource seeks to offer direction for those needing to provide updates to content, whether in a template for a theme, pattern, or a block over an entire site. Since each content type allows or disallows certain kind of synchronization, it's important to know what's possible before creating to make maintenance easier in the future.

Recommendations on managing updates

Establish early what content you expect to require updates

At a high level, it's important to recognize that not every piece of content can be updated across the entire site and that the method of creation greatly impacts what's possible. As a result, it's critical to spend time ahead of creation determining what you expect to need updates and to put that content in the appropriate format. This will make a huge difference in terms of future maintenance.

Embrace theme design at the block level

Block theme design requires a mindset shift from the previous approach of designing large sections of a theme and controlling them via updates. While a holistic view of a design is still important when creating a custom theme project, blocks require that themers approach design on a more atomic level. This means starting from the block itself, typically through theme.json customizations. **The goal is that each individual “atom” (i.e., block) can be moved around, edited, deleted, and put back together without the entire design falling apart.**

The more that you approach design at the block level, the less need there is to propagate updates to things like patterns and templates across the entire site. If the atomic pieces are in place, their layout should not matter.

Content types (and how to properly update them)

Blocks

How to manage block updates depends on the nature of the block itself. If the block depends on external data, then making it dynamic from start with the `render_callback` function is often a better choice as it provides more control. If the block's structure is expected to change over time, then starting with the static block that uses `save()` method defining a default output is the recommended approach. Over time, it's possible to go hybrid and include also the `render_callback` that can use the output from `save` as a fallback while processing an alternate output. Keep in mind that that flexibility and controls comes at the cost of additional processing during rendering. Another option is using static blocks that rely on managing updates with [block deprecations](#). This will require you to manually update exist blocks. Depending on

your needs and comfortability, either approach can work. **To get started on creating blocks and save time, [you can use the Create Block tool](#).**

Patterns

For content that you want updated later on, do not use patterns and instead rely on reusable blocks or template parts. Patterns cannot be updated after you insert one into your site. For context, you can think of Patterns as more like sample/example/starter content. While Patterns exposed in the Inserter might evolve over time, those changes won't be automatically applied to any current usage of the pattern. Once inserted, patterns become completely detached from the original pattern unlike Reusable block or Template Part block.

If needed, one potential workaround for patterns with custom styles is to add a class name to the wrapping block for a pattern. For example, the following adds a themeslug-special class to a Group block:

```
<!-- wp:group {"className": "themeslug-special"} -->
<div class="wp-block-group themeslug-special">
    <!-- Nested pattern blocks -->
</div>
<!-- /wp:group -->
```

It is not fool-proof because users can modify the class via the editor UI. However, because this setting is under the “Advanced” panel it is likely to stay intact in most instances. This gives theme authors some CSS control for some pattern types, allowing them to update existing uses. However, it does not prevent users from making massive alterations that cannot be updated.

Reusable blocks

As the name suggests, these blocks are inherently synced across your site. Keep in mind that there are currently limitations with relying on reusable blocks to handle certain updates since content, HTML structure, and styles will all stay in sync when updates happen. If you require more nuance than that, this is a key element to factor in and a dynamic block might be a better approach.

Template parts and templates

Because block themes allow users to directly edit templates and template parts, how changes are managed need to be adjusted in light of the greater access given to users. For context, when templates or template parts are changed by the user, the updated templates from the theme update don't show for the user. Only new users of the theme or users who have not yet edited a template are experiencing the updated template. If users haven't modified the files then the changes you make on the file system will be reflected on the user's sites – you just need to update the files and they'll get the changes. However if they have made changes to their templates then the only way you can update their templates is to:

- Revert all their changes
- Update the templates and template parts in the database

Generally speaking, if a user has made changes to templates then it's recommended to leave the templates as is, unless agreed upon with the user (ie in an agency setting).

One thing to be mindful of when updating templates is inserting references to new or different template parts. For example, the templates/page.html template could insert a parts/header.html

part in version 1.0 but change that reference to parts/header-alt.html in version 2.0. Some developers may see this as a “workaround” in instances where users modified the original header.html. However, this is likely to break a user’s customized design since the page.html template would no longer reference the correct part unless they also modified and saved the page template.

Likewise, it is generally poor practice to delete template parts in theme updates. In this scenario, users could create custom top-level templates that include a call to the part and expect it to continue existing.

[Resources](#)

- [Comparing Patterns, Template Parts, and Reusable Blocks](#)
- [Block deprecation](#)
- [Create Block tool](#)

First published

September 6, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Propagating updates for block types”](#)

[Previous Plugin Sidebar](#) [Previous: Plugin Sidebar](#)
[Next Themes](#) [Next: Themes](#)

Themes

In this article

Table of Contents

- [Types of themes](#)
 - [Classic theme](#)
 - [Block theme](#)
 - [Full site editing \(FSE\)](#)

[↑ Back to top](#)

The block editor provides a number of options for theme designers and developers, to interact with it, including theme-defined color settings, font size control, and much more.

Types of themes

Classic theme

In terms of block editor terminology this is any theme that defines its templates in the traditional .php file format, and that doesn't have an index.html format template in the /block-templates or /templates folders. A Classic theme has the ability to provide configuration and styling options to the block editor, and block content, via [Theme Supports](#), or by including a [theme.json](#) file. A theme does not have to be a Block theme in order to take advantage of some of the flexibility provided by the use of a theme.json file.

Block theme

This is any theme that has, at a minimum, an index.html format template in the /block-templates or /templates folders, and with templates provided in form of block content markup. While many Block themes will make use of a theme.json file to provide configuration and styling settings, a theme.json is not a requirement of Block themes. The advantage of Block themes is that the block editor can be used to edit all areas of the site: headers, footers, sidebars, etc.

Full site editing (FSE)

There isn't an FSE specific theme type. In WordPress > 5.9 FSE is enabled for any Block theme, ie. any theme that has an index.html format template in the /block-templates or /templates folders.

Contents

- [Global Settings \(theme.json\)](#)
- [Theme Support](#)

First published

April 26, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Themes”](#)

Global Settings & Styles (theme.json)

In this article

Table of Contents

- [Rationale](#)
 - [Settings for the block editor](#)
 - [Settings can be controlled per block](#)
 - [Styles are managed](#)
 - [CSS Custom Properties: presets & custom](#)
- [Specification](#)
 - [Version](#)
 - [Settings](#)
 - [Opt-in into UI controls](#)
 - [Styles](#)
 - [Top-level styles](#)
 - [Block styles](#)
 - [customTemplates](#)
 - [templateParts](#)
 - [patterns](#)
- [Developing with theme.json](#)
- [Frequently Asked Questions](#)
 - [The naming schema of CSS Custom Properties](#)
 - [Why using — as a separator?](#)
 - [How settings under “custom” create new CSS Custom Properties](#)
 - [Global Stylesheet](#)
 - [Specificity for link colors provided by the user](#)
 - [What is blockGap and how can I use it?](#)
 - [Why does it take so long to update the styles in the browser?](#)

[↑ Back to top](#)

WordPress 5.8 comes with [a new mechanism](#) to configure the editor that enables a finer-grained control and introduces the first step in managing styles for future WordPress releases: the `theme.json` file. Then `theme.json` [evolved to a v2](#) with WordPress 5.9 release. This page documents its format.

[Rationale](#)

The Block Editor API has evolved at different velocities and there are some growing pains, specially in areas that affect themes. Examples of this are: the ability to [control the editor programmatically](#), or [a block style system](#) that facilitates user, theme, and core style preferences.

This describes the current efforts to consolidate the various APIs related to styles into a single point – a `theme.json` file that should be located inside the root of the theme directory.

Settings for the block editor

Instead of the proliferation of theme support flags or alternative methods, the `theme.json` files provides a canonical way to define the settings of the block editor. These settings includes things like:

- What customization options should be made available or hidden from the user.
- What are the default colors, font sizes... available to the user.
- Defines the default layout of the editor (widths and available alignments).

Settings can be controlled per block

For more granularity, these settings also work at the block level in `theme.json`.

Examples of what can be achieved are:

- Use a particular preset for a block (e.g.: table) but the common one for the rest of blocks.
- Enable font size UI controls for all blocks but the headings block.
- etc.

Styles are managed

By using the `theme.json` file to set style properties in a structured way, the Block Editor can “manage” the CSS that comes from different origins (user, theme, and core CSS). For example, if a theme and a user set the font size for paragraphs, we only enqueue the style coming from the user and not the theme’s.

Some of the advantages are:

- Reduce the amount of CSS enqueued.
- Prevent specificity wars.

CSS Custom Properties: presets & custom

There are some areas of styling that would benefit from having shared values that can change across a site.

To address this need, we’ve started to experiment with CSS Custom Properties, aka CSS Variables, in some places:

- **Presets:** [color palettes](#), [font sizes](#), or [gradients](#) declared by the theme are converted to CSS Custom Properties and enqueued both the front-end and the editors.

Input	Output
{ "version": 2, "settings": { "color": { "palette": [{ "name": "Black", "slug": "black", "color": "#000000" }] } }	

```

        } ,
        {
            "name": "White",
            "slug": "white",
            "color": "#ffffff"
        }
    ]
}
}

body {
    --wp--preset--color--black: #000000;
    --wp--preset--color--white: #ffffff;
}

```

- **Custom properties:** there's also a mechanism to create your own CSS Custom Properties.

Input	Output

```

{
    "version": 2,
    "settings": {
        "custom": {
            "line-height": {
                "body": 1.7,
                "heading": 1.3
            }
        }
    }
}

body {
    --wp--custom--line-height--body: 1.7;
    --wp--custom--line-height--heading: 1.3;
}

```

Specification

This specification is the same for the three different origins that use this format: core, themes, and users. Themes can override core's defaults by creating a file called `theme.json`. Users, via the site editor, will also be able to override theme's or core's preferences via an user interface that is being worked on.

```
{
    "version": 2,
    "settings": {},
    "styles": {},
    "customTemplates": {},
    "templateParts": {}
}
```

Version

This field describes the format of the `theme.json` file. The current version is [v2](#), [introduced in WordPress 5.9](#). It also works with the current Gutenberg plugin.

If you have used [v1](#) previously, you don't need to update the version in the v1 file to v2, as it'll be [migrated](#) into v2 at runtime for you.

Settings

The Gutenberg plugin extends the settings available from WordPress 5.8, so they can be used with other WordPress versions and they go through a maturation process before being ported to core.

The tabs below show WordPress 5.8 supported settings and the ones supported by the Gutenberg plugin.

The settings section has the following structure:

WordPress	Gutenberg
-----------	-----------

```
{  
    "version": 2,  
    "settings": {  
        "border": {  
            "radius": false,  
            "color": false,  
            "style": false,  
            "width": false  
        },  
        "color": {  
            "custom": true,  
            "customDuotone": true,  
            "customGradient": true,  
            "duotone": [],  
            "gradients": [],  
            "link": false,  
            "palette": [],  
            "text": true,  
            "background": true,  
            "defaultGradients": true,  
            "defaultPalette": true  
        },  
        "custom": {},  
        "layout": {  
            "contentSize": "800px",  
            "wideSize": "1000px"  
        },  
        "spacing": {  
            "margin": false,  
            "padding": false,  
            "blockGap": null,  
            "units": [ "px", "em", "rem", "vh", "vw" ]  
        }  
    }  
}
```

```
        },
        "typography": {
            "customFontSize": true,
            "lineHeight": false,
            "dropCap": true,
            "fluid": false,
            "fontStyle": true,
            "fontWeight": true,
            "letterSpacing": true,
            "textDecoration": true,
            "textTransform": true,
            "fontSizes": [],
            "fontFamilies": []
        },
        "blocks": {
            "core/paragraph": {
                "color": {},
                "custom": {},
                "layout": {},
                "spacing": {},
                "typography": {}
            },
            "core/heading": {},
            "etc": {}
        }
    }
}

{
    "version": 2,
    "settings": {
        "appearanceTools": false,
        "border": {
            "color": false,
            "radius": false,
            "style": false,
            "width": false
        },
        "color": {
            "background": true,
            "custom": true,
            "customDuotone": true,
            "customGradient": true,
            "defaultGradients": true,
            "defaultPalette": true,
            "duotone": [],
            "gradients": [],
            "link": false,
            "palette": [],
            "text": true
        },
        "custom": {},
        "dimensions": {
            "aspectRatio": false,

```

```

        "minHeight": false,
    },
    "layout": {
        "contentSize": "800px",
        "wideSize": "1000px"
    },
    "spacing": {
        "blockGap": null,
        "margin": false,
        "padding": false,
        "customSpacingSize": true,
        "units": [ "px", "em", "rem", "vh", "vw" ],
        "spacingScale": {
            "operator": "*",
            "increment": 1.5,
            "steps": 7,
            "mediumStep": 1.5,
            "unit": "rem"
        },
        "spacingSizes": []
    },
    "typography": {
        "customFontSize": true,
        "dropCap": true,
        "fluid": false,
        "fontFamilies": [],
        "fontSizes": [],
        "fontStyle": true,
        "fontWeight": true,
        "letterSpacing": true,
        "lineHeight": false,
        "textColumns": false,
        "textDecoration": true,
        "textTransform": true
    },
    "blocks": {
        "core/paragraph": {
            "border": {},
            "color": {},
            "custom": {},
            "layout": {},
            "spacing": {},
            "typography": {}
        },
        "core/heading": {},
        "etc": {}
    }
}
}

```

Each block can configure any of these settings separately, providing a more fine-grained control over what exists via `add_theme_support`. The settings declared at the top-level affect to all blocks, unless a particular block overwrites it. It's a way to provide inheritance and configure all blocks at once.

Note, however, that not all settings are relevant for all blocks. The settings section provides an opt-in/opt-out mechanism for themes, but it's the block's responsibility to add support for the features that are relevant to it. For example, if a block doesn't implement the `dropCap` feature, a theme can't enable it for such a block through `theme.json`.

Opt-in into UI controls

There's one special setting property, `appearanceTools`, which is a boolean and its default value is false. Themes can use this setting to enable the following ones:

- `background: backgroundImage`
- `border: color, radius, style, width`
- `color: link`
- `dimensions: minHeight`
- `position: sticky`
- `spacing: blockGap, margin, padding`
- `typography: lineHeight`

Backward compatibility with `add_theme_support`

To retain backward compatibility, the existing `add_theme_support` declarations that configure the block editor are retrofitted in the proper categories for the top-level section. For example, if a theme uses `add_theme_support('disable-custom-colors')`, it'll be the same as setting `settings.color.custom` to `false`. If the `theme.json` contains any settings, these will take precedence over the values declared via `add_theme_support`. This is the complete list of equivalences:

add_theme_support	theme.json setting
<code>custom-line-height</code>	Set <code>typography.lineHeight</code> to <code>true</code> .
<code>custom-spacing</code>	Set <code>spacing.padding</code> to <code>true</code> .
<code>custom-units</code>	Provide the list of units via <code>spacing.units</code> .
<code>disable-custom-colors</code>	Set <code>color.custom</code> to <code>false</code> .
<code>disable-custom-font-sizes</code>	Set <code>typography.customFontSize</code> to <code>false</code> .
<code>disable-custom-gradients</code>	Set <code>color.customGradient</code> to <code>false</code> .
<code>editor-color-palette</code>	Provide the list of colors via <code>color.palette</code> .
<code>editor-font-sizes</code>	Provide the list of font size via <code>typography.fontSizes</code> .
<code>editor-gradient-presets</code>	Provide the list of gradients via <code>color.gradients</code> .
<code>appearance-tools</code>	Set <code>appearanceTools</code> to <code>true</code> .
<code>border</code>	Set <code>border: color, radius, style, width</code> to <code>true</code> .
<code>link-color</code>	Set <code>color.link</code> to <code>true</code> .

Presets

Presets are part of the settings section. They are values that are shown to the user via some UI controls. By defining them via `theme.json` the engine can do more for themes, such as automatically translate the preset name or enqueue the corresponding CSS classes and custom properties.

The following presets can be defined via `theme.json`:

- `color.duotone`: doesn't generate classes or custom properties.
- `color.gradients`: generates a single class and custom property per preset value.
- `color.palette`:
 - generates 3 classes per preset value: color, background-color, and border-color.
 - generates a single custom property per preset value.
- `spacing.spacingScale`: used to generate an array of spacing preset sizes for use with padding, margin, and gap settings.
 - `operator`: specifies how to calculate the steps with either * for multiplier, or + for sum.
 - `increment`: the amount to increment each step by. Core by default uses a 'perfect 5th' multiplier of 1.5.
 - `steps`: the number of steps to generate in the spacing scale. The default is 7. To prevent the generation of the spacing presets, and to disable the related UI, this can be set to 0.
 - `mediumStep`: the steps in the scale are generated descending and ascending from a medium step, so this should be the size value of the medium space, without the unit. The default medium step is 1.5rem so the mediumStep value is 1.5.
 - `unit`: the unit the scale uses, eg. px, rem, em, %. The default is rem.
- `spacing.spacingSizes`: themes can choose to include a static `spacing.spacingSizes` array of spacing preset sizes if they have a sequence of sizes that can't be generated via an increment or multiplier.
 - `name`: a human readable name for the size, eg. Small, Medium, Large.
 - `slug`: the machine readable name. In order to provide the best cross site/theme compatibility the slugs should be in the format, "10", "20", "30", "40", "50", "60", with "50" representing the Medium size value.
 - `size`: the size, including the unit, eg. 1.5rem. It is possible to include fluid values like clamp(2rem, 10vw, 20rem).
- `typography.fontSizes`: generates a single class and custom property per preset value.
- `typography.fontFamilies`: generates a single custom property per preset value.

The naming schema for the classes and the custom properties is as follows:

- Custom Properties: --wp--preset--{preset-category}--{preset-slug} such as --wp--preset--color--black
- Classes: .has-{preset-slug}-{preset-category} such as .has-black-color.

Input	Output
-------	--------

```
{  
    "version": 2,  
    "settings": {  
        "color": {  
            "duotone": [  
                "black",  
                "white"  
            ]  
        }  
    }  
}
```

```
        {
          "colors": [ "#000", "#FFF" ],
          "slug": "black-and-white",
          "name": "Black and White"
        }
      ],
      "gradients": [
        {
          "slug": "blush-bordeaux",
          "gradient": "linear-gradient(135deg,rgb(254,205,165) 0",
          "name": "Blush bordeaux"
        },
        {
          "slug": "blush-light-purple",
          "gradient": "linear-gradient(135deg,rgb(255,206,236) 0",
          "name": "Blush light purple"
        }
      ],
      "palette": [
        {
          "slug": "strong-magenta",
          "color": "#a156b4",
          "name": "Strong magenta"
        },
        {
          "slug": "very-dark-grey",
          "color": "rgb(131, 12, 8)",
          "name": "Very dark grey"
        }
      ]
    },
    "typography": {
      "fontFamilies": [
        {
          "fontFamily": "-apple-system,BlinkMacSystemFont,\"Segoe UI\",Ubuntu,Helvetica Neue, Helvetica, Arial, sans-serif",
          "slug": "system-font",
          "name": "System Font"
        },
        {
          "fontFamily": "Helvetica Neue, Helvetica, Arial, sans-serif",
          "slug": "helvetica-arial",
          "name": "Helvetica or Arial"
        }
      ],
      "fontSizes": [
        {
          "slug": "big",
          "size": 32,
          "name": "Big"
        },
        {
          "slug": "x-large",
          "size": 46,
          "name": "Large"
        }
      ]
    }
  }
}
```

```
        }
    ],
},
"spacing": {
    "spacingScale": {
        "operator": "*",
        "increment": 1.5,
        "steps": 7,
        "mediumStep": 1.5,
        "unit": "rem"
    },
    "spacingSizes": [
        {
            "slug": "40",
            "size": "1rem",
            "name": "Small"
        },
        {
            "slug": "50",
            "size": "1.5rem",
            "name": "Medium"
        },
        {
            "slug": "60",
            "size": "2rem",
            "name": "Large"
        }
    ]
},
"blocks": {
    "core/group": {
        "color": {
            "palette": [
                {
                    "slug": "black",
                    "color": "#000000",
                    "name": "Black"
                },
                {
                    "slug": "white",
                    "color": "#ffffff",
                    "name": "White"
                }
            ]
        }
    }
}
}

/* Top-level custom properties */
body {
    --wp--preset--color--strong-magenta: #a156b4;
    --wp--preset--color--very-dark-grey: #444;
```

```

--wp--preset--gradient--blush-bordeaux: linear-gradient( 135deg, rgb(
--wp--preset--gradient--blush-light-purple: linear-gradient( 135deg, r
--wp--preset--font-size--x-large: 46;
--wp--preset--font-size--big: 32;
--wp--preset--font-family--helvetica-arial: Helvetica Neue, Helvetica,
--wp--preset--font-family--system: -apple-system,BlinkMacSystemFont,\"
--wp--preset--spacing--20: 0.44rem;
--wp--preset--spacing--30: 0.67rem;
--wp--preset--spacing--40: 1rem;
--wp--preset--spacing--50: 1.5rem;
--wp--preset--spacing--60: 2.25rem;
--wp--preset--spacing--70: 3.38rem;
--wp--preset--spacing--80: 5.06rem;
}

/* Block-level custom properties (bounded to the group block) */
.wp-block-group {
    --wp--preset--color--black: #000000;
    --wp--preset--color--white: #ffffff;
}

/* Top-level classes */
.has-strong-magenta-color { color: #a156b4 !important; }
.has-strong-magenta-background-color { background-color: #a156b4 !importan
.has-strong-magenta-border-color { border-color: #a156b4 !important; }
.has-very-dark-grey-color { color: #444 !important; }
.has-very-dark-grey-background-color { background-color: #444 !important;
.has-very-dark-grey-border-color { border-color: #444 !important; }
.has-blush-bordeaux-background { background: linear-gradient( 135deg, rgb(
.has-blush-light-purple-background { background: linear-gradient( 135deg,
.has-big-font-size { font-size: 32; }
.has-normal-font-size { font-size: 16; }

/* Block-level classes (bounded to the group block) */
.wp-block-group.has-black-color { color: #a156b4 !important; }
.wp-block-group.has-black-background-color { background-color: #a156b4 !im
.wp-block-group.has-black-border-color { border-color: #a156b4 !important; }
.wp-block-group.has-white-color { color: #444 !important; }
.wp-block-group.has-white-background-color { background-color: #444 !imp
.wp-block-group.has-white-border-color { border-color: #444 !important; }

```

To maintain backward compatibility, the presets declared via `add_theme_support` will also generate the CSS Custom Properties. If the `theme.json` contains any presets, these will take precedence over the ones declared via `add_theme_support`.

Preset classes are attached to the content of a post by some user action. That's why the engine will add `!important` to these, because user styles should take precedence over theme styles.

Custom

In addition to create CSS Custom Properties for the presets, the `theme.json` also allows for themes to create their own, so they don't have to be enqueued separately. Any values declared

within the `custom` field will be transformed to CSS Custom Properties following this naming schema: `--wp--custom--<variable-name>`.

For example:

Input	Output
<pre>{ "version": 2, "settings": { "custom": { "baseFont": 16, "lineHeight": { "small": 1.2, "medium": 1.4, "large": 1.8 } }, "blocks": { "core/group": { "custom": { "baseFont": 32 } } } } body { --wp--custom--base-font: 16; --wp--custom--line-height--small: 1.2; --wp--custom--line-height--medium: 1.4; --wp--custom--line-height--large: 1.8; } .wp-block-group { --wp--custom--base-font: 32; }</pre>	

Note that the name of the variable is created by adding `--` in between each nesting level and `camelCase` fields are transformed to `kebab-case`.

Settings examples

- Enable custom colors only for the paragraph block:

Input	Output
<pre>{ "version": 2, "settings": { "color": { "custom": false }, "blocks": { "core/paragraph": { "color": { "custom": true } } } } body { --wp--custom--color--base: red; } .wp-block-paragraph { --wp--custom--color--base: green; }</pre>	<pre>body { --wp--custom--color--base: red; } .wp-block-paragraph { --wp--custom--color--base: green; }</pre>

```
        "custom": true
    }
}
}
}
}
```

- Disable border radius for the button block:

```
{  
    "version": 2,  
    "settings": {  
        "blocks": {  
            "core/button": {  
                "border": {  
                    "radius": false  
                }  
            }  
        }  
    }  
}
```

- Provide the group block a different palette than the rest:

```
{  
  "version": 2,  
  "settings": {  
    "color": {  
      "palette": [  
        {  
          "slug": "black",  
          "color": "#000000",  
          "name": "Black"  
        },  
        {  
          "slug": "white",  
          "color": "#FFFFFF",  
          "name": "White"  
        },  
        {  
          "slug": "red",  
          "color": "#FF0000",  
          "name": "Red"  
        },  
        {  
          "slug": "green",  
          "color": "#00FF00",  
          "name": "Green"  
        },  
        {  
          "slug": "blue",  
          "color": "#0000FF",  
          "name": "Blue"  
        }  
      ]  
    }  
  }  
}
```

```

        ]
    },
    "blocks": {
        "core/group": {
            "color": {
                "palette": [
                    {
                        "slug": "black",
                        "color": "#000000",
                        "name": "Black"
                    },
                    {
                        "slug": "white",
                        "color": "#FFF",
                        "name": "White"
                    }
                ]
            }
        }
    }
}

```

Styles

The Gutenberg plugin extends the styles available from WordPress 5.8, so they can be used with other WordPress versions and they go through a maturation process before being ported to core.

The tabs below show WordPress 5.8 supported styles and the ones supported by the Gutenberg plugin.

Each block declares which style properties it exposes via the [block supports mechanism](#). The support declarations are used to automatically generate the UI controls for the block in the editor. Themes can use any style property via the `theme.json` for any block — it's the theme's responsibility to verify that it works properly according to the block markup, etc.

WordPress	Gutenberg
-----------	-----------

```
{
    "version": 2,
    "styles": {
        "border": {
            "radius": "value",
            "color": "value",
            "style": "value",
            "width": "value"
        },
        "filter": {
            "duotone": "value"
        },
        "color": {
            "background": "value",
            "gradient": "value",
            "text": "value"
        }
    }
}
```

```
},
"spacing": {
    "blockGap": "value",
    "margin": {
        "top": "value",
        "right": "value",
        "bottom": "value",
        "left": "value",
    },
    "padding": {
        "top": "value",
        "right": "value",
        "bottom": "value",
        "left": "value"
    }
},
"typography": {
    "fontSize": "value",
    "fontStyle": "value",
    "fontWeight": "value",
    "letterSpacing": "value",
    "lineHeight": "value",
    "textDecoration": "value",
    "textTransform": "value"
},
"elements": {
    "link": {
        "border": {},
        "color": {},
        "spacing": {},
        "typography": {}
    },
    "h1": {},
    "h2": {},
    "h3": {},
    "h4": {},
    "h5": {},
    "h6": {}
},
"blocks": {
    "core/group": {
        "border": {},
        "color": {},
        "spacing": {},
        "typography": {},
        "elements": {
            "link": {},
            "h1": {},
            "h2": {},
            "h3": {},
            "h4": {},
            "h5": {},
            "h6": {}
        }
    }
}
```

```
        },
        "etc": {}
    }
}

{
    "version": 2,
    "styles": {
        "border": {
            "color": "value",
            "radius": "value",
            "style": "value",
            "width": "value"
        },
        "color": {
            "background": "value",
            "gradient": "value",
            "text": "value"
        },
        "dimensions": {
            "aspectRatio": "value",
            "minHeight": "value"
        },
        "filter": {
            "duotone": "value"
        },
        "spacing": {
            "blockGap": "value",
            "margin": {
                "top": "value",
                "right": "value",
                "bottom": "value",
                "left": "value"
            },
            "padding": {
                "top": "value",
                "right": "value",
                "bottom": "value",
                "left": "value"
            }
        },
        "typography": {
            "fontFamily": "value",
            "fontSize": "value",
            "fontStyle": "value",
            "fontWeight": "value",
            "letterSpacing": "value",
            "lineHeight": "value",
            "textColumns": "value",
            "textDecoration": "value",
            "textTransform": "value"
        },
        "elements": {
```

```

    "link": {
      "border": {},
      "color": {},
      "spacing": {},
      "typography": {}
    },
    "h1": {},
    "h2": {},
    "h3": {},
    "h4": {},
    "h5": {},
    "h6": {},
    "heading": {},
    "button": {},
    "caption": {}
  },
  "blocks": {
    "core/group": {
      "border": {},
      "color": {},
      "dimensions": {},
      "spacing": {},
      "typography": {},
      "elements": {
        "link": {},
        "h1": {},
        "h2": {},
        "h3": {},
        "h4": {},
        "h5": {},
        "h6": {}
      }
    },
    "etc": {}
  }
}
}

```

Top-level styles

Styles found at the top-level will be enqueueued using the `body` selector.

Input	Output
-------	--------

```
{
  "version": 1,
  "styles": {
    "color": {
      "text": "var(--wp--preset--color--primary)"
    }
  }
}
```

```
body {  
    color: var( --wp--preset--color--primary );  
}
```

Block styles

Styles found within a block will be enqueued using the block selector.

By default, the block selector is generated based on its name such as `.wp-block-<blockname-without-namespace>`. For example, `.wp-block-group` for the core/group block. There are some blocks that want to opt-out from this default behavior. They can do so by explicitly telling the system which selector to use for them via the `__experimentalSelector` key within the `supports` section of its `block.json` file. Note that the block needs to be registered server-side for the `__experimentalSelector` field to be available to the style engine.

Input	Output
<pre>{ "version": 1, "styles": { "color": { "text": "var(--wp--preset--color--primary)" }, "blocks": { "core/paragraph": { "color": { "text": "var(--wp--preset--color--secondary)" } }, "core/group": { "color": { "text": "var(--wp--preset--color--tertiary)" } } } } } body { color: var(--wp--preset--color--primary); } p { /* The core/paragraph opts out from the default behaviour and uses p a color: var(--wp--preset--color--secondary); } .wp-block-group { color: var(--wp--preset--color--tertiary); }</pre>	<pre>body { color: var(--wp--preset--color--primary); } p { /* The core/paragraph opts out from the default behaviour and uses p a color: var(--wp--preset--color--secondary); } .wp-block-group { color: var(--wp--preset--color--tertiary); }</pre>

Referencing a style

A block can be styled using a reference to a root level style. This feature is supported by Gutenberg.

If you register a background color for the root using styles.color.background:

```
"styles": {  
    "color": {  
        "background": "var(--wp--preset--color--primary)"  
    }  
}
```

You can use ref: "styles.color.background" to re-use the style for a block:

```
{  
    "color": {  
        "text": { "ref": "styles.color.background" }  
    }  
}
```

Element styles

In addition to top-level and block-level styles, there's the concept of elements that can be used in both places. There's a closed set of them:

Supported by Gutenberg:

- button: maps to the wp-element-button CSS class. Also maps to wp-block-button_link for backwards compatibility.
- caption: maps to the .wp-element-caption, .wp-block-audio figcaption, .wp-block-embed figcaption, .wp-block-gallery figcaption, .wp-block-image figcaption, .wp-block-table figcaption, .wp-block-video figcaption CSS classes.
- heading: maps to all headings, the h1 to h6 CSS selectors.

Supported by WordPress:

- h1: maps to the h1 CSS selector.
- h2: maps to the h2 CSS selector.
- h3: maps to the h3 CSS selector.
- h4: maps to the h4 CSS selector.
- h5: maps to the h5 CSS selector.
- h6: maps to the h6 CSS selector.
- link: maps to the a CSS selector.

If they're found in the top-level the element selector will be used. If they're found within a block, the selector to be used will be the element's appended to the corresponding block.

Input	Output
-------	--------

```
{  
    "version": 1,  
    "styles": {  
        "typography": {
```

```
        "fontSize": "var(--wp--preset--font-size--normal)"
    },
    "elements": {
        "h1": {
            "typography": {
                "fontSize": "var(--wp--preset--font-size--huge)"
            }
        },
        "h2": {
            "typography": {
                "fontSize": "var(--wp--preset--font-size--big)"
            }
        },
        "h3": {
            "typography": {
                "fontSize": "var(--wp--preset--font-size--medium)"
            }
        }
    },
    "blocks": {
        "core/group": {
            "elements": {
                "h2": {
                    "typography": {
                        "fontSize": "var(--wp--preset--font-size--small)"
                    }
                },
                "h3": {
                    "typography": {
                        "fontSize": "var(--wp--preset--font-size--small)"
                    }
                }
            }
        }
    }
}

body {
    font-size: var( --wp--preset--font-size--normal );
}
h1 {
    font-size: var( --wp--preset--font-size--huge );
}
h2 {
    font-size: var( --wp--preset--font-size--big );
}
h3 {
    font-size: var( --wp--preset--font-size--medium );
}
.wp-block-group h2 {
    font-size: var( --wp--preset--font-size--small );
}
.wp-block-group h3 {
```

```
    font-size: var( --wp--preset--font-size--smaller );
}
```

Element pseudo selectors

Pseudo selectors :hover, :focus, :visited, :active, :link, :any-link are supported by Gutenberg.

```
"elements": {
    "link": {
        "color": {
            "text": "green"
        },
        ":hover": {
            "color": {
                "text": "hotpink"
            }
        }
    }
}
```

Variations

A block can have a “style variation”, as defined per the [block.json specification](#). Theme authors can define the style attributes for an existing style variation using the theme.json file. Styles for unregistered style variations will be ignored.

Note that variations are a “block concept”, they only exist bound to blocks. The theme.json specification respects that distinction by only allowing variations at the block-level but not at the top-level. It’s also worth highlighting that only variations defined in the block.json file of the block are considered “registered”: so far, the style variations added via `register_block_style` or in the client are ignored, see [this issue](#) for more information.

For example, this is how to provide styles for the existing plain variation for the core/quote block:

```
{
    "version": 2,
    "styles": {
        "blocks": {
            "core/quote": {
                "variations": {
                    "plain": {
                        "color": {
                            "background": "red"
                        }
                    }
                }
            }
        }
    }
}
```

The resulting CSS output is this:

```
.wp-block-quote.is-style-plain {
    background-color: red;
}
```

customTemplates

Supported in WordPress from version 5.9.

Within this field themes can list the custom templates present in the `templates` folder. For example, for a custom template named `my-custom-template.html`, the `theme.json` can declare what post types can use it and what's the title to show the user:

- name: mandatory.
- title: mandatory, translatable.
- postTypes: optional, only applies to the page by default.

```
{
    "version": 2,
    "customTemplates": [
        {
            "name": "my-custom-template",
            "title": "The template title",
            "postTypes": [
                "page",
                "post",
                "my-cpt"
            ]
        }
    ]
}
```

templateParts

Supported in WordPress from version 5.9.

Within this field themes can list the template parts present in the `parts` folder. For example, for a template part named `my-template-part.html`, the `theme.json` can declare the area term for the template part entity which is responsible for rendering the corresponding block variation (Header block, Footer block, etc.) in the editor. Defining this area term in the json will allow the setting to persist across all uses of that template part entity, as opposed to a block attribute that would only affect one block. Defining area as a block attribute is not recommended as this is only used ‘behind the scenes’ to aid in bridging the gap between placeholder flows and entity creation.

Currently block variations exist for “header” and “footer” values of the area term, any other values and template parts not defined in the json will default to the general template part block. Variations will be denoted by specific icons within the editor’s interface, will default to the corresponding semantic HTML element for the wrapper (this can also be overridden by the `tagName` attribute set on the template part block), and will contextualize the template part allowing more custom flows in future editor improvements.

- name: mandatory.
- title: optional, translatable.
- area: optional, will be set to `uncategorized` by default and trigger no block variation.

```
{
  "version": 2,
  "templateParts": [
    {
      "name": "my-template-part",
      "title": "Header",
      "area": "header"
    }
  ]
}
```

[patterns](#)

Supported in WordPress from version 6.0 using [version 2](#) of theme.json.

Within this field themes can list patterns to register from [Pattern Directory](#). The patterns field is an array of pattern slugs from the Pattern Directory. Pattern slugs can be extracted by the url in single pattern view at the Pattern Directory. For example in this url <https://wordpress.org/patterns/partner-logos> the slug is partner-logos`.

```
{
  "version": 2,
  "patterns": [ "short-text-surrounded-by-round-images", "partner-logos" ]
}
```

[Developing with theme.json](#)

It can be difficult to remember the theme.json settings and properties while you develop, so a JSON schema was created to help. The schema is available at <https://schemas.wp.org/trunk/theme.json>

Code editors can pick up the schema and can provide help like tooltips, autocomplete, or schema validation in the editor. To use the schema in Visual Studio Code, add "\$schema": "<https://schemas.wp.org/trunk/theme.json>" to the beginning of your theme.json file.

```
theme.json M ●
1 {
2
3     "version": "1",
4     "settings": {
5         "colour": {
6             "custom": false,
7             "palette": [
8                 {
9                     "name": "black",
10                    "slug": "black",
11                    "color": "#000000"
12                },
13                {
14                    "name": "white",
15                    "slug": "white",
16                    "color": "#FFFFFF"
17                }
18            ],
19            "font-size": {
20                "size": "1em"
21            }
22        }
23    }
24}
```

Frequently Asked Questions

The naming schema of CSS Custom Properties

One thing you may have noticed is the naming schema used for the CSS Custom Properties the system creates, including the use of double hyphen, --, to separate the different “concepts”. Take the following examples.

Presets such as --wp--preset--color--black can be divided into the following chunks:

- --wp: prefix to namespace the CSS variable.
- preset: indicates is a CSS variable that belongs to the presets.
- color: indicates which preset category the variable belongs to. It can be color, font-size, gradients.
- black: the slug of the particular preset value.

Custom properties such as `--wp--custom--line-height--body`, which can be divided into the following chunks:

- `--wp`: prefix to namespace the CSS variable.
- `custom`: indicates is a “free-form” CSS variable created by the theme.
- `line-height--body`: the result of converting the “custom” object keys into a string.

The `--` as a separator has two functions:

- Readability, for human understanding. It can be thought as similar to the BEM naming schema, it separates “categories”.
- Parsability, for machine understanding. Using a defined structure allows machines to understand the meaning of the property `--wp--preset--color--black`: it’s a value bounded to the color preset whose slug is “black”, which then gives us room to do more things with them.

Why using — as a separator?

We could have used any other separator, such as a single `-`.

However, that’d have been problematic, as it’d have been impossible to tell how `--wp--custom-line-height-template-header` should be converted back into an object, unless we force theme authors not to use `-` in their variable names.

By reserving `--` as a category separator and let theme authors use `-` for word-boundaries, the naming is clearer: `--wp--custom--line-height--template--header`.

How settings under “custom” create new CSS Custom Properties

The algorithm to create CSS Variables out of the settings under the “custom” key works this way:

This is for clarity, but also because we want a mechanism to parse back a variable name such `--wp--custom--line-height--body` to its object form in `theme.json`. We use the same separation for presets.

For example:

Input	Output
{ "version": 2, "settings": { "custom": { "lineHeight": { "body": 1.7 }, "font-primary": "-apple-system, BlinkMacSystemFont, 'Segoe UI" } } } body { --wp--custom--line-height--body: 1.7; }	

```
--wp--custom--font-primary: "-apple-system, BlinkMacSystemFont, 'Segoe UI", "Lucida Grande", sans-serif";
```

A few notes about this process:

- camelCased keys are transformed into its kebab-case form, as to follow the CSS property naming schema. Example: `lineHeight` is transformed into `line-height`.
- Keys at different depth levels are separated by `--`. That's why `line-height` and `body` are separated by `--`.
- You shouldn't use `--` in the names of the keys within the `custom` object. Example, **don't do this**:

```
{
  "version": 2,
  "settings": {
    "custom": {
      "line--height": { // DO NOT DO THIS
        "body": 1.7
      }
    }
  }
}
```

Global Stylesheet

In WordPress 5.8, the CSS for some of the presets defined by WordPress (font sizes, colors, and gradients) was loaded twice for most themes: in the block-library stylesheet plus in the global stylesheet. Additionally, there were slight differences in the CSS in both places.

In WordPress 5.9 release, CSS of presets are consolidated into the global stylesheet, that is now loaded for all themes. Each preset value generates a single CSS Custom Property and a class, as in:

```
/* CSS Custom Properties for the preset values */
body {
  --wp--preset--<PRESET_TYPE>--<PRESET_SLUG>: <DEFAULT_VALUE>;
  --wp--preset--color--pale-pink: #f78da7;
  --wp--preset--font-size--large: 36px;
  /* etc. */
}

/* CSS classes for the preset values */
.has-<PRESET_SLUG>-<PRESET_TYPE> { ... }
.has-pale-pink-color { color: var(--wp--preset--color--pale-pink) !important;
.has-large-font-size { font-size: var(--wp--preset--font-size--large) !important;
```

For themes to override the default values they can use the `theme.json` and provide the same slug. Themes that do not use a `theme.json` can still override the default values by enqueueing some CSS that sets the corresponding CSS Custom Property.

Example (sets a new value for the default large font size):

```
body {  
    --wp--preset--font-size--large: <NEW_VALUE>;  
}
```

[Specificity for link colors provided by the user](#)

In v1, when a user selected a link color for a specific block we attached a class to that block in the form of `.wp-element-<ID>` and then enqueued the following style:

```
.wp-element-<ID> a { color: <USER_COLOR_VALUE> !important; }
```

While this preserved user preferences at all times, the specificity was too strong and conflicted with some blocks with legit uses of an HTML element that shouldn't be considered links. To [address this issue](#), in WordPress 5.9 release, the `!important` was removed and updated the corresponding blocks to style the `a` elements with a specificity higher than the user link color, which now is:

```
.wp-element-<ID> a { color: <USER_COLOR_VALUE>; }
```

As a result of this change, it's now the block author and theme author's responsibility to make sure the user choices are respected at all times and that the link color provided by the user (specificity 011) is not overridden.

[What is blockGap and how can I use it?](#)

For blocks that contain inner blocks, such as Group, Columns, Buttons, and Social Icons, `blockGap` controls the spacing between inner blocks. For `blockGap` to work, the block must also opt in to the [layout block support](#), which provides layout styles that can be adjusted via the block spacing controls. Depending on the layout of the block, the `blockGap` value will be output as either a vertical margin or a `gap` value. In the editor, the control for the `blockGap` value is called *Block spacing*, located in the Dimensions panel.

```
{  
    "version": 2,  
    "settings": {  
        "spacing": {  
            "blockGap": true,  
        }  
    },  
    "styles": {  
        "spacing": {  
            "blockGap": "1.5rem"  
        }  
    }  
}
```

The setting for `blockGap` is either a boolean or `null` value and is `null` by default. This allows an extra level of control over style output. The `settings.spacing.blockGap` setting in a `theme.json` file accepts the following values:

- `true`: Opt into displaying *Block spacing* controls in the editor UI and output `blockGap` styles.

- `false`: Opt out of displaying *Block spacing* controls in the editor UI, with `blockGap` styles stored in `theme.json` still being rendered. This allows themes to use `blockGap` values without allowing users to make changes within the editor.
- `null` (default): Opt out of displaying *Block spacing* controls, *and* prevent the output of `blockGap` styles.

The value defined for the root `styles.spacing.blockGap` style is also output as a CSS property, named `--wp--style--block-gap`.

Why does it take so long to update the styles in the browser?

When you are actively developing with `theme.json` you may notice it takes 30+ seconds for your changes to show up in the browser, this is because `theme.json` is cached. To remove this caching issue, set either [WP_DEBUG](#) or [SCRIPT_DEBUG](#) to ‘true’ in your [wp-config.php](#). This tells WordPress to skip the cache and always use fresh data.

First published

December 28, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Global Settings & Styles \(theme.json\)](#)”

[Previous Themes](#) [Previous: Themes](#)

[Next Theme Support](#) [Next: Theme Support](#)

Theme Support

In this article

Table of Contents

- [Opt-in features](#)
- [Default block styles](#)
 - [Wide Alignment:](#)
 - [Wide Alignments and Floats](#)
 - [Block Color Palettes](#)
 - [Block Gradient Presets](#)
 - [Block Font Sizes](#)
 - [Disabling custom font sizes](#)
 - [Disabling custom colors in block Color Palettes](#)
 - [Disabling custom gradients](#)
 - [Disabling base layout styles](#)
 - [Supporting custom line heights](#)
 - [Support custom units](#)
 - [Disabling the default block patterns.](#)

- [Editor styles](#)
 - [Enqueuing the editor style](#)
 - [Basic colors](#)
 - [Changing the width of the editor](#)
- [Responsive embedded content](#)
- [Spacing control](#)
- [Link color control](#)
- [Appearance Tools](#)
- [Border](#)
- [Link color](#)
- [Block Based Template Parts](#)

[↑ Back to top](#)

The new Blocks include baseline support in all themes, enhancements to opt-in to and the ability to extend and customize.

There are a few new concepts to consider when building themes:

- **Editor Color Palette** – A default set of colors is provided, but themes can register their own and optionally lock users into picking from the defined palette.
- **Editor Text Size Palette** – A default set of sizes is provided, but themes can register their own and optionally lock users into picking from preselected sizes.
- **Responsive Embeds** – Themes must opt-in to responsive embeds.
- **Frontend & Editor Styles** – To get the most out of blocks, theme authors will want to make sure Core styles look good and opt-in, or write their own styles to best fit their theme.
- **Block Tools** – Themes can opt-in to several block tools like line height, custom units.
- **Core Block Patterns** – Themes can opt-out of the default block patterns.

By default, blocks provide their styles to enable basic support for blocks in themes without any change. They also [provide opt-in opinionated styles](#). Themes can add/override these styles, or they can provide no styles at all, and rely fully on what the blocks provide.

Some advanced block features require opt-in support in the theme itself as it's difficult for the block to provide these styles, they may require some architecting of the theme itself, in order to work well.

To opt-in for one of these features, call `add_theme_support` in the `functions.php` file of the theme. For example:

```
function mytheme_setup_theme_supported_features() {
    add_theme_support( 'editor-color-palette', array(
        array(
            'name'  => esc_attr__( 'strong magenta', 'themeLangDomain' ),
            'slug'  => 'strong-magenta',
            'color' => '#a156b4',
        ),
        array(
            'name'  => esc_attr__( 'light grayish magenta', 'themeLangDomain' ),
            'slug'  => 'light-grayish-magenta',
            'color' => '#d0a5db',
        ),
        array(
            'name'  => esc_attr__( 'very light gray', 'themeLangDomain' ),
            'slug'  => 'very-light-gray',
            'color' => '#e0e0e0',
        ),
    ),
    array(
        'name'  => esc_attr__( 'light gray', 'themeLangDomain' ),
        'slug'  => 'light-gray',
        'color' => '#c0c0c0',
    ),
)
```

```

        'slug'  => 'very-light-gray',
        'color' => '#eee',
    ),
    array(
        'name'  => esc_attr__( 'very dark gray', 'themeLangDomain' ),
        'slug'  => 'very-dark-gray',
        'color' => '#444',
    ),
) );
)
);
}

add_action( 'after_setup_theme', 'mytheme_setup_theme_supported_features'

```

Opt-in features

Default block styles

Core blocks include default structural styles. These are loaded in both the editor and the front end by default. An example of these styles is the CSS that powers the columns block. Without these rules, the block would result in a broken layout containing no columns at all.

The block editor allows themes to opt-in to slightly more opinionated styles for the front end. An example of these styles is the default color bar to the left of blockquotes. If you'd like to use these opinionated styles in your theme, add theme support for `wp-block-styles`:

```
add_theme_support( 'wp-block-styles' );
```

You can see the CSS that will be included in the [block library theme file](#).

Wide Alignment:

Some blocks such as the image block have the possibility to define a “wide” or “full” alignment by adding the corresponding classname to the block’s wrapper (`alignwide` or `alignfull`). A theme can opt-in for this feature by calling:

```
add_theme_support( 'align-wide' );
```

For more information about this function, see [the developer docs on add_theme_support\(\)](#).

Wide Alignments and Floats

It can be difficult to create a responsive layout that accommodates wide images, a sidebar, a centered column, and floated elements that stay within that centered column.

The block editor adds additional markup to floated images to make styling them easier.

Here's the markup for an `Image` with a caption:

```
<figure class="wp-block-image">
    
    <figcaption>Short image caption.</figcaption>
</figure>
```

Here's the markup for a left-floated image:

```
<div class="wp-block-image">
  <figure class="alignleft">
    
    <figcaption>Short image caption.</figcaption>
  </figure>
</div>
```

Here's an example [codepen](#) using the above markup to achieve a responsive layout that features a sidebar, wide images, and floated elements with bounded captions.

[Block Color Palettes](#)

Different blocks have the possibility of customizing colors. The block editor provides a default palette, but a theme can overwrite it and provide its own:

```
add_theme_support( 'editor-color-palette', array(
  array(
    'name'  => esc_attr__( 'strong magenta', 'themeLangDomain' ),
    'slug'  => 'strong-magenta',
    'color' => '#a156b4',
  ),
  array(
    'name'  => esc_attr__( 'light grayish magenta', 'themeLangDomain' ),
    'slug'  => 'light-grayish-magenta',
    'color' => '#d0a5db',
  ),
  array(
    'name'  => esc_attr__( 'very light gray', 'themeLangDomain' ),
    'slug'  => 'very-light-gray',
    'color' => '#eee',
  ),
  array(
    'name'  => esc_attr__( 'very dark gray', 'themeLangDomain' ),
    'slug'  => 'very-dark-gray',
    'color' => '#444',
  ),
) );
```

`name` is a human-readable label (demonstrated above) that appears in the tooltip and provides a meaningful description of the color to users. It is especially important for those who rely on screen readers or would otherwise have difficulty perceiving the color. `slug` is a unique identifier for the color and is used to generate the CSS classes used by the block editor color palette. `color` is the hexadecimal code to specify the color.

Some colors change dynamically — such as “Primary” and “Secondary” color — such as in the Twenty Nineteen theme and cannot be described programmatically. In spite of that, it is still advisable to provide meaningful names for at least the default values when applicable.

The colors will be shown in order on the palette, and there's no limit to how many can be specified.

Themes are responsible for creating the classes that apply the colors in different contexts. Core blocks use “color”, “background-color”, and “border-color” contexts. So to correctly apply “strong magenta” to all contexts of core blocks a theme should implement the classes itself. The class name is built appending ‘has-‘, followed by the class name *using* kebab case and ending with the context name.

```
.has-strong-magenta-color {  
    color: #a156b4;  
}  
  
.has-strong-magenta-background-color {  
    background-color: #a156b4;  
}  
  
.has-strong-magenta-border-color {  
    border-color: #a156b4;  
}
```

Starting in WordPress 5.9, to override color values defined by core, themes without a theme .json have to set their values via CSS Custom Properties instead of providing the classes. The CSS Custom Properties use the following naming --wp--preset--color--<slug>. See more info in [this devnote](#). For example:

```
:root {  
    --wp--preset--color--cyan-bluish-gray: <new_value>;  
    --wp--preset--color--pale-pink: <new_value>;  
}
```

Block Gradient Presets

Different blocks have the possibility of selecting from a list of predefined gradients. The block editor provides a default gradient presets, but a theme can overwrite them and provide its own:

```
add_theme_support(  
    'editor-gradient-presets',  
    array(  
        array(  
            'name'      => esc_attr__( 'Vivid cyan blue to vivid purple', ' ',  
            'gradient'  => 'linear-gradient(135deg,rgba(6,147,227,1) 0%,rgb(6,147,227,1)',  
            'slug'      => 'vivid-cyan-blue-to-vivid-purple'  
        ),  
        array(  
            'name'      => esc_attr__( 'Vivid green cyan to vivid cyan blue', ' ',  
            'gradient'  => 'linear-gradient(135deg,rgba(0,208,132,1) 0%,rgb(0,208,132,1)',  
            'slug'      => 'vivid-green-cyan-to-vivid-cyan-blue',  
        ),  
        array(  
            'name'      => esc_attr__( 'Light green cyan to vivid green cyan', ' ',  
            'gradient'  => 'linear-gradient(135deg,rgb(122,220,180) 0%,rgb(122,220,180)',  
            'slug'      => 'light-green-cyan-to-vivid-green-cyan',  
        ),  
        array(  
            'name'      => esc_attr__( 'Luminous vivid amber to luminous violet', ' ',  
            'gradient'  => 'linear-gradient(135deg,rgba(252,185,0,1) 0%,rgba(252,185,0,1)',  
            'slug'      => 'luminous-vivid-amber-to-luminous-violet',  
        )  
    )  
);
```

```

        'slug'      => 'luminous-vivid-amber-to-luminous-vivid-orange',
    ),
    array(
        'name'      => esc_attr__( 'Luminous vivid orange to vivid red',
        'gradient' => 'linear-gradient(135deg,rgba(255,105,0,1) 0%,rgb(255,105,0,1) 100%)',
        'slug'       => 'luminous-vivid-orange-to-vivid-red',
    ),
),
);

```

`name` is a human-readable label (demonstrated above) that appears in the tooltip and provides a meaningful description of the gradient to users. It is especially important for those who rely on screen readers or would otherwise have difficulty perceiving the color. `gradient` is a CSS value of a gradient applied to a background-image of the block. Details of valid gradient types can be found in the [mozilla documentation](#). `slug` is a unique identifier for the gradient and is used to generate the CSS classes used by the block editor.

Themes are responsible for creating the classes that apply the gradients. So to correctly apply “Vivid cyan blue to vivid purple” a theme should implement the following class:

```

.has-vivid-cyan-to-vivid-purple-gradient-background {
    background: linear-gradient(
        135deg,
        rgba( 6, 147, 227, 1 ) 0%,
        rgb( 155, 81, 224 ) 100%
    );
}

```

Starting in WordPress 5.9, to override gradient values defined by core, themes without a `theme.json` have to set their values via CSS Custom Properties instead of providing the classes. The CSS Custom Properties use the following naming `--wp--preset--gradient--<slug>`. See more info in [this devnote](#). For example:

```

:root {
    --wp--preset--gradient--vivid-cyan-blue-to-vivid-purple: <new_value>;
    --wp--preset--gradient--light-green-cyan-to-vivid-green-cyan: <new_value>;
}

```

Block Font Sizes

Blocks may allow the user to configure the font sizes they use, e.g., the paragraph block. The block provides a default set of font sizes, but a theme can overwrite it and provide its own:

```

add_theme_support( 'editor-font-sizes', array(
    array(
        'name' => esc_attr__( 'Small', 'themeLangDomain' ),
        'size' => 12,
        'slug' => 'small'
    ),
    array(
        'name' => esc_attr__( 'Regular', 'themeLangDomain' ),
        'size' => 16,
        'slug' => 'regular'
    ),
)
);

```

```

array(
    'name' => esc_attr__( 'Large', 'themeLangDomain' ),
    'size' => 36,
    'slug' => 'large'
),
array(
    'name' => esc_attr__( 'Huge', 'themeLangDomain' ),
    'size' => 50,
    'slug' => 'huge'
)
);
)
);

```

The font sizes are rendered on the font size picker in the order themes provide them.

Themes are responsible for creating the classes that apply the correct font size styles. The class name is built appending ‘has-‘, followed by the font size name *using* kebab case and ending with `-font-size`.

As an example for the regular font size, a theme may provide the following class.

```
.has-regular-font-size {
    font-size: 16px;
}
```

Note: The slugs ‘default’ and ‘custom’ are reserved and cannot be used by themes.

Starting in WordPress 5.9, to override font size values defined by core, themes without a theme.json have to set their values via CSS Custom Properties instead of providing the classes. The CSS Custom Properties use the following naming `--wp--preset--font-size--<slug>`. See more info in [this devnote](#). For example:

```
:root {
    --wp--preset--font-size--small: <new_value>;
    --wp--preset--font-size--large: <new_value>;
}
```

Disabling custom font sizes

Themes can disable the ability to set custom font sizes with the following code:

```
add_theme_support( 'disable-custom-font-sizes' );
```

When set, users will be restricted to the default sizes provided in the block editor or the sizes provided via the `editor-font-sizes` theme support setting.

Disabling custom colors in block Color Palettes

By default, the color palette offered to blocks allows the user to select a custom color different from the editor or theme default colors.

Themes can disable this feature using:

```
add_theme_support( 'disable-custom-colors' );
```

This flag will make sure users are only able to choose colors from the `editor-color-palette` the theme provided or from the editor default colors if the theme did not provide one.

Disabling custom gradients

Themes can disable the ability to set a custom gradient with the following code:

```
add_theme_support( 'disable-custom-gradients' );
```

When set, users will be restricted to the default gradients provided in the block editor or the gradients provided via the `editor-gradient-presets` theme support setting.

Disabling base layout styles

Note: Since WordPress 6.1.

Themes can opt out of generated block layout styles that provide default structural styles for core blocks including Group, Columns, Buttons, and Social Icons. By using the following code, these themes commit to providing their own structural styling, as using this feature will result in core blocks displaying incorrectly in both the editor and site frontend:

```
add_theme_support( 'disable-layout-styles' );
```

For themes looking to customize `blockGap` styles or block spacing, see [the developer docs on Global Settings & Styles](#).

Supporting custom line heights

Some blocks like paragraph and headings support customizing the line height. Themes can enable support for this feature with the following code:

```
add_theme_support( 'custom-line-height' );
```

Support custom units

In addition to pixels, users can use other units to define sizes, paddings... The available units are: px, em, rem, vh, vw. Themes can disable support for this feature with the following code:

```
add_theme_support( 'custom-units', array() );
```

Themes can also filter the available custom units.

```
add_theme_support( 'custom-units', 'rem', 'em' );
```

Disabling the default block patterns.

WordPress comes with a number of block patterns built-in, themes can opt-out of the bundled patterns and provide their own set using the following code:

```
remove_theme_support( 'core-block-patterns' );
```

Editor styles

The block editor supports the theme's [editor styles](#), however it works a little differently than in the classic editor.

In the classic editor, the editor stylesheet is loaded directly into the iframe of the WYSIWYG editor, with no changes. The block editor, however, doesn't use iframes. To make sure your styles are applied only to the content of the editor, we automatically transform your editor styles by selectively rewriting or adjusting certain CSS selectors. This also allows the block editor to leverage your editor style in block variation previews.

For example, if you write `body { ... }` in your editor style, this is rewritten to `.editor-styles-wrapper { ... }`. This also means that you should *not* target any of the editor class names directly.

Because it works a little differently, you need to opt-in to this by adding an extra snippet to your theme, in addition to the `add_editor_style` function:

```
add_theme_support( 'editor-styles' );
```

You shouldn't need to change your editor styles too much; most themes can add the snippet above and get similar results in the classic editor and inside the block editor.

Enqueuing the editor style

Use the `add_editor_style` function to enqueue and load CSS on the editor screen. For the classic editor, this was the only function needed to add style to the editor. For the new block editor, you first need to `add_theme_support('editor-styles')`; mentioned above.

```
add_editor_style( 'style-editor.css' );
```

Adding that to your `functions.php` file will add the stylesheet `style-editor.css` to the queue of stylesheets to be loaded in the editor.

Basic colors

You can style the editor like any other webpage. Here's how to change the background color and the font color to blue:

```
/* Add this to your `style-editor.css` file */
body {
    background-color: #d3ebf3;
    color: #00005d;
}
```

Changing the width of the editor

To change the main column width of the editor, add the following CSS to `style-editor.css`:

```
/* Main column width */
.wp-block {
    max-width: 720px;
}
```

```
/* Width of "wide" blocks */
.wp-block[data-align='wide'] {
    max-width: 1080px;
}

/* Width of "full-wide" blocks */
.wp-block[data-align='full'] {
    max-width: none;
}
```

You can use those editor widths to match those in your theme. You can use any CSS width unit, including % or px.

Further reading: [Applying Styles with Stylesheets](#).

Responsive embedded content

The embed blocks automatically apply styles to embedded content to reflect the aspect ratio of content that is embedded in an iFrame. A block styled with the aspect ratio responsive styles would look like:

```
<figure class="wp-embed-aspect-16-9 wp-has-aspect-ratio">...</figure>
```

To make the content resize and keep its aspect ratio, the <body> element needs the wp-embed-responsive class. This is not set by default, and requires the theme to opt in to the responsive-embeds feature:

```
add_theme_support( 'responsive-embeds' );
```

Spacing control

Some blocks can have padding controls. This is off by default, and requires the theme to opt in by declaring support:

```
add_theme_support( 'custom-spacing' );
```

Link color control

Link support has been made stable as part of WordPress 5.8. It's `false` by default and themes can enable it via the [theme.json file](#):

```
{
    "version": 1,
    "settings": {
        "color": {
            "link": true
        }
    }
}
```

Alternatively, with the Gutenberg plugin active, the old legacy support `add_theme_support('experimental-link-color')` would also work. This fallback would be removed when the Gutenberg plugin requires WordPress 5.9 as the minimum version.

When the user sets the link color of a block, a new style will be added in the form of:

```
.wp-elements-<uuid> a {  
    color: <link-color> !important;  
}
```

where

- `<uuid>` is a random number
- `<link-color>` is either `var(--wp--preset--color--slug)` (if the user selected a preset value) or a raw color value (if the user selected a custom value)

The block will get attached the class `.wp-elements-<uuid>`.

[Appearance Tools](#)

Use this setting to enable the following Global Styles settings:

- `border`: color, radius, style, width
- `color`: link
- `spacing`: blockGap, margin, padding
- `typography`: lineHeight
- `dimensions`: aspectRatio, minHeight
- `position`: sticky

```
add_theme_support( 'appearance-tools' );
```

[Border](#)

Use this to enable all border settings:

```
add_theme_support( 'border' );
```

[Link color](#)

Use this to enable the link color setting:

```
add_theme_support( 'link-color' );
```

[Block Based Template Parts](#)

Block Based Template parts allow administrators to edit parts of the site using blocks. This is off by default, and requires the theme to opt in by declaring support:

```
add_theme_support( 'block-template-parts' );
```

This feature is only relevant for non block based themes, as block based themes already support block based template parts as part of the site editor.

The standalone template part editor does not allow editors to create new, or delete existing template parts. This is because the theme manually needs to include the template part in the PHP template.

You can find out more about block based template parts in the [themes handbook block template and template parts section](#).

First published

April 26, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Theme Support”](#)

[Previous Global Settings & Styles \(theme.json\)](#) [Previous: Global Settings & Styles \(theme.json\)](#)
[Next Thunks in Core-Data](#) [Next: Thunks in Core-Data](#)

Thunks in Core-Data

In this article

Table of Contents

- [Why are thunks useful?](#)
 - [Thunks have access to the store helpers](#)
 - [Thunks can be async](#)
- [Thunks API](#)
 - [select](#)
 - [resolveSelect](#)
 - [dispatch](#)
 - [registry](#)

[↑ Back to top](#)

[Gutenberg 11.6](#) added support for *thunks*. You can think of thunks as functions that can be dispatched:

```
// actions.js
export const myThunkAction = () => ( { select, dispatch } ) => {
    return "I'm a thunk! I can be dispatched, use selectors, and even disp
};
```

Why are thunks useful?

Thunks [expand the meaning of what a Redux action is](#). Before thunks, actions were purely functional and could only return and yield data. Common use cases such as interacting with the store or requesting API data from an action required using a separate [control](#). You would often see code like:

```
export function* saveRecordAction( id ) {
  const record = yield controls.select( 'current-store', 'getRecord', id );
  yield { type: 'BEFORE_SAVE', id, record };
  const results = yield controls.fetch({ url: 'https://...', method: 'POST' });
  yield { type: 'AFTER_SAVE', id, results };
  return results;
}

const controls = {
  select: // ...,
  fetch: // ...,
};
```

Side effects like store operations and fetch functions would be implemented outside of the action. Thunks provide an alternative to this approach. They allow you to use side effects inline, like this:

```
export const saveRecordAction = ( id ) => async ({ select, dispatch }) =>
  const record = select( 'current-store', 'getRecord', id );
  dispatch({ type: 'BEFORE_SAVE', id, record });
  const response = await fetch({ url: 'https://...', method: 'POST', data: record });
  const results = await response.json();
  dispatch({ type: 'AFTER_SAVE', id, results });
  return results;
}
```

This removes the need to implement separate controls.

Thunks have access to the store helpers

Let's take a look at an example from Gutenberg core. Prior to thunks, the `toggleFeature` action from the `@wordpress/interface` package was implemented like this:

```
export function* toggleFeature( scope, featureName ) {
  const currentValue = yield controls.select(
    interfaceStoreName,
    'isFeatureActive',
    scope,
    featureName
  );

  yield controls.dispatch(
    interfaceStoreName,
    'setFeatureValue',
    scope,
    featureName,
```

```

        ! currentValue
    );
}

```

Controls were the only way to `dispatch` actions and `select` data from the store.

With thunks, there is a cleaner way. This is how `toggleFeature` is implemented now:

```

export function toggleFeature( scope, featureName ) {
    return function ( { select, dispatch } ) {
        const currentValue = select.isFeatureActive( scope, featureName );
        dispatch.setFeatureValue( scope, featureName, ! currentValue );
    };
}

```

Thanks to the `select` and `dispatch` arguments, thunks may use the store directly without the need for generators and controls.

Thunks can be async

Imagine a simple React app that allows you to set the temperature on a thermostat. It only has one input and one button. Clicking the button dispatches a `saveTemperatureToAPI` action with the value from the input.

If we used controls to save the temperature, the store definition would look like below:

```

const store = wp.data.createReduxStore( 'my-store' , {
    actions: {
        saveTemperatureToAPI: function*( temperature ) {
            const result = yield { type: 'FETCH_JSON', url: 'https://...' ,
                return result;
            }
        },
        controls: {
            async FETCH_JSON( action ) {
                const response = await window.fetch( action.url, {
                    method: action.method,
                    body: JSON.stringify( action.data ),
                } );
                return response.json();
            }
        },
        // reducers, selectors, ...
    }
);

```

While the code is reasonably straightforward, there is a level of indirection. The `saveTemperatureToAPI` action does not talk directly to the API, but has to go through the `FETCH_JSON` control.

Let's see how this indirection can be removed with thunks:

```

const store = wp.data.createReduxStore( 'my-store' , {
    actions: {
        saveTemperatureToAPI: ( temperature ) => async () => {

```

```

        const response = await window.fetch( 'https://...' , {
            method: 'POST',
            body: JSON.stringify( { temperature } ),
        } );
        return await response.json();
    }
},
// reducers, selectors, ...
} );

```

That's pretty cool! What's even better is that resolvers are supported as well:

```

const store = wp.data.createReduxStore( 'my-store' , {
    // ...
    selectors: {
        getTemperature: ( state ) => state.temperature
    },
    resolvers: {
        getTemperature: () => async ( { dispatch } ) => {
            const response = await window.fetch( 'https://...' );
            const result = await response.json();
            dispatch.receiveCurrentTemperature( result.temperature );
        }
    },
    // ...
} );

```

Support for thunks is included by default in every data store, just like the (now legacy) support for generators and controls.

Thunks API

A thunk receives a single object argument with the following keys:

select

An object containing the store's selectors pre-bound to state, which means you don't need to provide the state, only the additional arguments. `select` triggers the related resolvers, if any, but does not wait for them to finish. It just returns the current value even if it's null.

If a selector is part of the public API, it's available as a method on the `select` object:

```

const thunk = () => ( { select } ) => {
    // select is an object of the store's selectors, pre-bound to current
    const temperature = select.getTemperature();
}

```

Since not all selectors are exposed on the store, `select` doubles as a function that supports passing a selector as an argument:

```

const thunk = () => ( { select } ) => {
    // select supports private selectors:
}

```

```
    const doubleTemperature = select( ( temperature ) => temperature * 2 )
}
```

resolveSelect

`resolveSelect` is the same as `select`, except it returns a promise that resolves with the value provided by the related resolver.

```
const thunk = () => ( { resolveSelect } ) => {
  const temperature = await resolveSelect.getTemperature();
}
```

dispatch

An object containing the store's actions

If an action is part of the public API, it's available as a method on the `dispatch` object:

```
const thunk = () => ( { dispatch } ) => {
  // dispatch is an object of the store's actions:
  const temperature = await dispatch.retrieveTemperature();
}
```

Since not all actions are exposed on the store, `dispatch` doubles as a function that supports passing a Redux action as an argument:

```
const thunk = () => async ( { dispatch } ) => {
  // dispatch is also a function accepting inline actions:
  dispatch({ type: 'SET_TEMPERATURE', temperature: result.value });

  // thunks are interchangeable with actions
  dispatch( updateTemperature( 100 ) );

  // Thunks may be async, too. When they are, dispatch returns a promise
  await dispatch( () => window.fetch( /* ... */ ) );
}
```

registry

A registry provides access to other stores through its `dispatch`, `select`, and `resolveSelect` methods.

These are very similar to the ones described above, with a slight twist. Calling `registry.select(storeName)` returns a function returning an object of selectors from `storeName`. This comes handy when you need to interact with another store. For example:

```
const thunk = () => ( { registry } ) => {
  const error = registry.select( 'core' ).getLastEntitySaveError( 'root',
    /* ... */
}
```

First published

October 29, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Thunks in Core-Data”](#)

[Previous Theme Support](#) [Previous: Theme Support](#)

[Next Widgets](#) [Next: Widgets](#)

Widgets

[↑ Back to top](#)

The Gutenberg plugin replaces the Widgets Editor screen in WP Admin with a new screen based on the WordPress block editor.

Contents

- [Widgets Block Editor overview](#)
- [Restoring the old Widgets Editor](#)
- [Ensuring compatibility with the Legacy Widget block](#)

First published

May 8, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Widgets”](#)

[Previous Thunks in Core-Data](#) [Previous: Thunks in Core-Data](#)

[Next Widgets Block Editor overview](#) [Next: Widgets Block Editor overview](#)

Widgets Block Editor overview

In this article

Table of Contents

- [Widgets Block Editor](#)
 - [Customizer Widgets Block Editor](#)
- [Compatibility](#)

[↑ Back to top](#)

Widgets Block Editor

The new Widgets Editor is a WordPress feature which upgrades widget areas to allow using blocks alongside widgets. It offers a new widget management experience built using the familiar WordPress block editor.

You can access the new Widgets Editor by navigating to Appearance → Widgets or Appearance → Customize → Widgets and choose a widget area.

The Widgets Block Editor allows you to insert blocks and widgets into any of the [Widget Areas or Sidebars](#) defined by the site's active theme, via a standalone editor or through the Customizer.

Customizer Widgets Block Editor

The new Widgets Editor also replaces the Widgets section in the Customizer with the new block-based editor.

You can access the Customizer Widgets Block Editor by navigating to Appearance → Customize, selecting Widgets, and then selecting a Widget Area.

Using the new Widgets Editor through the Customizer goes beyond inserting blocks and widgets into a selected Widget Area, making use of the live preview of the changes, to the right of the editor, and of all the other Customizer specific features such as scheduling and sharing changes.

Compatibility

Widgets that were added to a Widget Area before the new Widgets Editor will continue to work – via the Legacy Widget block.

The Legacy Widget block is the compatibility mechanism which allows us to edit and preview changes to a classic widget within the new block based Widgets Editor.

Any third party widgets registered by plugins can still be inserted in widget areas by adding and setting them up through a Legacy Widget block.

The Widgets Editor stores blocks using an underlying “Block” widget that is invisible to the user. This means that plugins and themes will continue to work normally, and that the Widgets Block Editor can be disabled without any data loss.

Themes may disable the Widgets Block Editor using
`remove_theme_support('widgets-block-editor').`

Users may disable the Widgets Block Editor by installing the [Classic Widgets plugin](#).

First published

May 8, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Widgets Block Editor overview](#)

[Previous Widgets](#) [Previous: Widgets](#)

[Next Restoring the classic Widgets Editor](#) [Next: Restoring the classic Widgets Editor](#)

Restoring the classic Widgets Editor

In this article

Table of Contents

- [Using remove_theme_support](#)
- [Using the Classic Widgets plugin](#)
- [Using a filter](#)

[↑ Back to top](#)

There are several ways to disable the new Widgets Block Editor.

[Using remove theme support](#)

Themes may disable the Widgets Block Editor by calling `remove_theme_support('widgets-block-editor')`.

For example, a theme may have the following PHP code in `functions.php`.

```
function example_theme_support() {  
    remove_theme_support( 'widgets-block-editor' );  
}  
add_action( 'after_setup_theme', 'example_theme_support' );
```

[Using the Classic Widgets plugin](#)

End users may disable the Widgets Block Editor by installing and activating the [Classic Widgets plugin](#).

With this plugin installed, the Widgets Block Editor can be toggled on and off by deactivating and activating the plugin.

[Using a filter](#)

The `use_widgets_block_editor` filter controls whether or not the Widgets Block Editor is enabled.

For example, a site administrator may include the following PHP code in a mu-plugin to disable the Widgets Block Editor.

```
add_filter( 'use_widgets_block_editor', '__return_false' );
```

For more advanced uses, you may supply your own function. In this example, the Widgets Block Editor is disabled for a specific user.

```
function example_use_widgets_block_editor( $use_widgets_block_editor ) {
    if ( 123 === get_current_user_id() ) {
        return false;
    }
    return $use_widgets_block_editor;
}
add_filter( 'use_widgets_block_editor', 'example_use_widgets_block_editor' );
```

First published

May 8, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Restoring the classic Widgets Editor](#)

[Previous Widgets Block Editor overview](#) [Previous: Widgets Block Editor overview](#)
[Next About the Legacy Widget block](#) [Next: About the Legacy Widget block](#)

About the Legacy Widget block

In this article

Table of Contents

- [Compatibility with the Legacy Widget block](#)
 - [The widget-added event](#)
 - [Displaying “No preview available.”](#)
 - [Allowing migration to a block](#)
- [Using the Legacy Widget block in other block editors \(Advanced\)](#)

[↑ Back to top](#)

The Legacy Widget block allows users to add, edit and preview third party widgets that are registered by plugins and widgets that were added using the classic Widgets Editor.

Third party widgets can be added by inserting a Legacy Widget block using the block inserter and selecting the widget from the block's dropdown.

Third party widgets may also be added by searching for the name of the widget in the block inserter and selecting the widget. A variation of the Legacy Widget block will be inserted.

Compatibility with the Legacy Widget block

The widget-added event

The Legacy Widget block will display the widget's form in a way similar to the Customizer, and so is compatible with most third party widgets.

If the widget uses JavaScript in its form, it is important that events are added to the DOM after the 'widget-added' jQuery event is triggered on document.

For example, a widget might want to show a “Password” field when the “Change password” checkbox is checked.

```
( function ( $ ) {
    $( document ).on( 'widget-added', function ( $event, $control ) {
        $control.find( '.change-password' ).on( 'change', function () {
            var isChecked = $( this ).prop( 'checked' );
            $control.find( '.password' ).toggleClass( 'hidden', ! isChecked );
        } );
    } );
} )( jQuery );
```

Note that all of the widget's event handlers are added in the `widget-added` callback.

Displaying “No preview available.”

The Legacy Widget block will display a preview of the widget when the Legacy Widget block is not selected.

A “No preview available.” message is automatically shown by the Legacy Widget block when the widget's `widget()` function does not render anything or only renders empty HTML elements.

Widgets may take advantage of this by returning early from `widget()` when a preview should not be displayed.

```
class ExampleWidget extends WP_Widget {
    ...
    public function widget( $instance ) {
        if ( ! isset( $instance['name'] ) ) {
            // Name is required, so display nothing if we don't have it.
            return;
        }
    ?>
    <h3>Name: <?php echo esc_html( $instance['name'] ); ?></h3>
    ...
    <?php
}
...
}
```

Allowing migration to a block

You can allow users to easily migrate a Legacy Widget block containing a specific widget to a block or multiple blocks. This allows plugin authors to phase out their widgets in favour of blocks which are more intuitive and can be used in more places.

The following steps show how to do this.

1) Display the widget's instance in the REST API

First, we need to tell WordPress that it is OK to display your widget's instance array in the REST API.

This can be safely done if:

- You know that all of the values stored by your widget in `$instance` can be represented as JSON; and
- You know that your widget does not store any private data in `$instance` that should be kept hidden from users that have permission to customize the site.

If it is safe to do so, then include a widget option named `show_instance_in_rest` with its value set to `true` when registering your widget.

```
class ExampleWidget extends WP_Widget {  
    ...  
    /**  
     * Sets up the widget  
     */  
    public function __construct() {  
        $widget_ops = array(  
            // ...other options here  
            'show_instance_in_rest' => true,  
            // ...other options here  
        );  
        parent::__construct( 'example_widget', 'ExampleWidget', $widget_ops  
    }  
    ...  
}
```

This allows the block editor and other REST API clients to see your widget's instance array by accessing `instance.raw` in the REST API response.

Note that [versions of WordPress prior to 5.8.0 allowed you to enable this feature by setting `\$show_instance_in_rest` to `true`](#) in the class that extends `WP_Widget`.

```
class ExampleWidget extends WP_Widget {  
    ...  
    public $show_instance_in_rest = true;  
    ...  
}
```

This is now deprecated in favour of the widget option method.

2) Add a block transform

Now, we can define a [block transform](#) which tells the block editor what to replace the Legacy Widget block containing your widget with.

This is done by adding JavaScript code to your block's definition. In this example, we define a transform that turns a widget with ID 'example_widget' into a block with name 'example/block'.

```
transforms: {
  from: [
    {
      type: 'block',
      blocks: [ 'core/legacy-widget' ],
      isMatch: ( { idBase, instance } ) => {
        if ( ! instance?.raw ) {
          // Can't transform if raw instance is not shown in RES
          return false;
        }
        return idBase === 'example_widget';
      },
      transform: ( { instance } ) => {
        return createBlock( 'example/block', {
          name: instance.raw.name,
        });
      },
    },
  ],
},
```

3) Hide the widget from the Legacy Widget block

As a final touch, we can tell the Legacy Widget block to hide your widget from the “Select widget” dropdown and from the block inserter. This encourages users to use the block that replaces your widget.

This can be done using the `widget_types_to_hide_from_legacy_widget_block` filter.

```
function hide_example_widget( $widget_types ) {
  $widget_types[] = 'example_widget';
  return $widget_types;
}
add_filter( 'widget_types_to_hide_from_legacy_widget_block', 'hide_example'
```

[Using the Legacy Widget block in other block editors](#) (Advanced)

You may optionally allow the Legacy Widget block in other block editors such as the WordPress post editor. This is not enabled by default.

First, ensure that any styles and scripts required by the legacy widgets are loaded onto the page. A convenient way of doing this is to manually perform all

of the hooks that ordinarily run when a user browses to the widgets WP Admin screen.

```
add_action( 'admin_print_styles', function() {
    if ( get_current_screen()->is_block_editor() ) {
        do_action( 'admin_print_styles-widgets.php' );
    }
} );
add_action( 'admin_print_scripts', function() {
    if ( get_current_screen()->is_block_editor() ) {
        do_action( 'load-widgets.php' );
        do_action( 'widgets.php' );
        do_action( 'sidebar_admin_setup' );
        do_action( 'admin_print_scripts-widgets.php' );
    }
} );
add_action( 'admin_print_footer_scripts', function() {
    if ( get_current_screen()->is_block_editor() ) {
        do_action( 'admin_print_footer_scripts-widgets.php' );
    }
} );
add_action( 'admin_footer', function() {
    if ( get_current_screen()->is_block_editor() ) {
        do_action( 'admin_footer-widgets.php' );
    }
} );
```

Then, register the Legacy Widget block using `registerLegacyWidgetBlock` which is defined in the `@wordpress/widgets` package.

```
add_action( 'enqueue_block_editor_assets', function() {
    wp_enqueue_script( 'wp-widgets' );
    wp_add_inline_script( 'wp-widgets', 'wp.widgets.registerLegacyWidgetBl
} );
```

First published

May 8, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: About the Legacy Widget block](#)

[Previous](#) [Restoring the classic Widgets Editor](#) [Previous: Restoring the classic Widgets Editor](#)
[Next Reference Guides](#) [Next: Reference Guides](#)

Reference Guides

In this article

Table of Contents

- [Block API Reference](#)
- [Hooks Reference](#)
- [SlotFills Reference](#)
- [Theme.json Reference](#)
- [RichText Reference](#)
- [Component Reference](#)
- [Package Reference](#)
- [Data Module Reference](#)

[↑ Back to top](#)

[Block API Reference](#)

- [Annotations](#)
- [API Versions](#)
- [Attributes](#)
- [Context](#)
- [Deprecation](#)
- [Edit and Save](#)
- [Patterns](#)
- [Registration](#)
- [Supports](#)
- [Templates](#)
- [Transformations](#)
- [Metadata](#)
- [Variations](#)

[Hooks Reference](#)

- [Block Filters](#)
- [Editor Hooks](#)
- [i18n Hooks](#)
- [Parser Hooks](#)
- [Autocomplete](#)
- [Global Styles Hooks](#)

[SlotFills Reference](#)

- [MainDashboardButton](#)
- [PluginBlockSettingsMenuItem](#)
- [PluginDocumentSettingPanel](#)
- [PluginMoreMenuItem](#)
- [PluginPostPublishPanel](#)

- [PluginPostStatusInfo](#)
- [PluginPrePublishPanel](#)
- [PluginSidebar](#)
- [PluginSidebarMoreMenuItem](#)

Theme.json Reference

- [Version 2 \(living reference\)](#)
- [Version 1](#)
- [Migrating to Newer Versions](#)

RichText Reference

Component Reference

Package Reference

Data Module Reference

- [**core:** WordPress Core Data](#)
 - [core/annotations:](#) Annotations
 - [core/block-directory:](#) Block directory
 - [core/block-editor:](#) The Block Editor's Data
 - [core/blocks:](#) Block Types Data
 - [core/customize-widgets:](#) Customize Widgets
 - [core/edit-post:](#) The Editor's UI Data
 - [core/edit-site:](#) Edit Site
 - [core/edit-widgets:](#) Edit Widgets
 - [core/editor:](#) The Post Editor's Data
 - [core/keyboard-shortcuts:](#) The Keyboard Shortcuts Data
 - [core/notices:](#) Notices Data
 - [core/nux:](#) The NUX (New User Experience) Data
 - [core/preferences:](#) Preferences
 - [core/reusable-blocks:](#) Reusable blocks
 - [core/rich-text:](#) Rich Text
 - [core/viewport:](#) The Viewport Data

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Reference Guides”](#)

Block API Reference

[↑ Back to top](#)

Blocks are the fundamental element of the editor. They are the primary way in which plugins and themes can register their own functionality and extend the capabilities of the editor.

The following sections will walk you through the existing block APIs:

- [Annotations](#)
- [API Versions](#)
- [Attributes](#)
- [Context](#)
- [Deprecation](#)
- [Edit and Save](#)
- [Metadata in block.json](#)
- [Patterns](#)
- [Registration](#)
- [Selectors](#)
- [Styles](#)
- [Supports](#)
- [Transformations](#)
- [Templates](#)
- [Variations](#)

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Block API Reference”](#)

[Previous Reference Guides](#) [Previous: Reference Guides](#)
[Next Annotations](#) [Next: Annotations](#)

Annotations

In this article

[Table of Contents](#)

- [API](#)
- [Block annotation](#)
- [Text annotation](#)

[↑ Back to top](#)

Note: This API is experimental, that means it is subject to non-backward compatible changes or removal in any future version.

Annotations are a way to highlight a specific piece in a post created with the block editor. Examples of this include commenting on a piece of text and spellchecking. Both can use the annotations API to mark a piece of text.

API

To see the API for yourself the easiest way is to have a block that is at least 200 characters long without formatting and putting the following in the console:

```
wp.data.dispatch( 'core/annotations' ).addAnnotation( {
    source: 'my-annotations-plugin',
    blockClientId: wp.data.select( 'core/block-editor' ).getBlockOrder()[0].clientID,
    richTextIdentifier: 'content',
    range: {
        start: 50,
        end: 100,
    },
} );
```

The start and the end of the range should be calculated based only on the text of the relevant RichText. For example, in the following HTML position 0 will refer to the position before the capital S:

```
<strong>Strong text</strong>
```

To help with determining the correct positions, the `wp.richText.create` method can be used. This will split a piece of HTML into text and formats.

All available properties can be found in the API documentation of the `addAnnotation` action.

The property `richTextIdentifier` is the identifier of the RichText instance the annotation applies to. This is necessary because blocks may have multiple rich text instances that are used to manage data for different attributes, so you need to pass this in order to highlight text within the correct one.

For example the Paragraph block only has a single RichText instance, with the identifier `content`. The quote block type has 2 RichText instances, so if you wish to highlight text in the citation, you need to pass `citation` as the `richTextIdentifier` when adding an annotation. To target the quote content, you need to use the identifier `value`. Refer to the source code of the block type to find the correct identifier.

Block annotation

It is also possible to annotate a block completely. In that case just provide the `selector` property and set it to `block`. The default `selector` is `range`, which can be used for text annotation.

```
wp.data.dispatch( 'core/annotations' ).addAnnotation( {
    source: 'my-annotations-plugin',
    blockClientId: wp.data.select( 'core/block-editor' ).getBlockOrder()[0].clientId,
    selector: 'block',
} );
```

This doesn't provide any styling out of the box, so you have to provide some CSS to make sure your annotation is shown:

```
.is-annotated-by-my-annotations-plugin {
    outline: 1px solid black;
}
```

[Text annotation](#)

The text annotation is controlled by the `start` and `end` properties. Simple `start` and `end` properties don't work for HTML, so these properties are assumed to be offsets within the rich-text internal structure. For simplicity you can think about this as if all HTML would be stripped out and then you calculate the `start` and the `end` of the annotation.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Annotations](#)

[Previous Block API Reference](#) [Previous: Block API Reference](#)

[Next API Versions](#) [Next: API Versions](#)

API Versions

In this article

Table of Contents

- [Version 3 \(>= WordPress 6.3\)](#)
- [Version 2 \(>= WordPress 5.6\)](#)
- [Version 1](#)

[↑ Back to top](#)

This document lists the changes made between the different API versions.

Version 3 (>= WordPress 6.3)

- The post editor will be iframed if all registered blocks have a Block API version 3 or higher and the editor has no classic meta boxes below the blocks. Adding version 3 support means that the block should work inside an iframe, though the block may still be rendered outside the iframe if not all blocks support version 3.

Version 2 (>= WordPress 5.6)

- To render the block element wrapper for the block's `edit` implementation, the block author must use the `useBlockProps()` hook.
- The generated class names and styles are no longer added automatically to the saved markup for static blocks when `save` is processed. To include them, the block author must explicitly use `useBlockProps.save()` and add to their block wrapper.

Version 1

Initial version.

First published

April 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: API Versions”](#)

[Previous Annotations](#) [Previous: Annotations](#)

[Next Attributes](#) [Next: Attributes](#)

Attributes

In this article

Table of Contents

- [Type validation](#)
- [Enum validation](#)
- [Value source](#)
 - [attribute source](#)
 - [text source](#)
 - [html source](#)
 - [query source](#)
 - [Meta source \(deprecated\)](#)
- [Default value](#)

[↑ Back to top](#)

Block attributes provide information about the data stored by a block. For example, rich content, a list of image URLs, a background colour, 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 s  
  )  
}
```

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"} -->

<!-- /block:your-block -->
```

If an attributes change 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
    
</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

        
</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>
  

  <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>
  

  <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>
  

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

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 `img` element in the block's markup.

Saved content:

```
<div>
  
  
</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:

```
{% JSX %}

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'  
  }  
}
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Attributes”](#)

[Previous API Versions](#) [Previous: API Versions](#)

[Next Context](#) [Next: Context](#)

Context

In this article

Table of Contents

- [Defining block context](#)
 - [Providing block context](#)
 - [Consuming block context](#)
- [Using block context](#)
 - [JavaScript](#)
 - [PHP](#)

- [Example](#)

[↑ Back to top](#)

Block context is a feature which enables ancestor blocks to provide values which can be consumed by descendent blocks within its own hierarchy. Those descendent blocks can inherit these values without resorting to hard-coded values and without an explicit awareness of the block which provides those values.

This is especially useful in full-site editing where, for example, the contents of a block may depend on the context of the post in which it is displayed. A blogroll template may show excerpts of many different posts. Using block context, there can still be one single “Post Excerpt” block which displays the contents of the post based on an inherited post ID.

If you are familiar with [React Context](#), block context adopts many of the same ideas. In fact, the client-side block editor implementation of block context is a very simple application of React Context. Block context is also supported in server-side `render_callback` implementations, demonstrated in the examples below.

[**Defining block context**](#)

Block context is defined in the registered settings of a block. A block can provide a context value, or consume a value it seeks to inherit.

[**Providing block context**](#)

A block can provide a context value by assigning a `providesContext` property in its registered settings. This is an object which maps a context name to one of the block’s own attribute. The value corresponding to that attribute value is made available to descendent blocks and can be referenced by the same context name. Currently, block context only supports values derived from the block’s own attributes. This could be enhanced in the future to support additional sources of context values.

```
attributes: {
    recordId: {
        type: 'number',
    },
},
providesContext: {
    'my-plugin/recordId': 'recordId',
},
```

For complete example, refer to the section below.

Include a namespace

As seen in the above example, it is recommended that you include a namespace as part of the name of the context key so as to avoid potential conflicts with other plugins or default context values provided by WordPress. The context namespace should be specific to your plugin, and in most cases can be the same as used in the name of the block itself.

Consuming block context

A block can inherit a context value from an ancestor provider by assigning a `usesContext` property in its registered settings. This should be assigned as an array of the context names the block seeks to inherit.

```
registerBlockType('my-plugin/record-title', {
    title: 'Record Title',
    category: 'widgets',
    usesContext: ['my-plugin/recordId'],
```

Using block context

Once a block has defined the context it seeks to inherit, this can be accessed in the implementation of `edit` (JavaScript) and `render_callback` (PHP). It is provided as an object (JavaScript) or associative array (PHP) of the context values which have been defined for the block. Note that a context value will only be made available if the block explicitly defines a desire to inherit that value.

Note: Block Context is not available to save.

JavaScript

```
registerBlockType('my-plugin/record-title', {
    edit({ context }) {
        return 'The record ID: ' + context['my-plugin/recordId'];
    },
});
```

PHP

A block's context values are available from the `context` property of the `$block` argument passed as the third argument to the `render_callback` function.

```
register_block_type( 'my-plugin/record-title', array(
    'render_callback' => function( $attributes, $content, $block ) {
        return 'The current record ID is: ' . $block->context['my-plugin/r
    },
) );
```

Example

1. Create record block.

```
npm init @wordpress/block --namespace my-plugin record
cd record
```

1. Edit `src/index.js`. Insert `recordId` attribute and `providesContext` property in `registerBlockType` function and add registration of `record-title` block at the bottom.

```
registerBlockType( 'my-plugin/record', {
    // ... cut ...

    attributes: {
        recordId: {
            type: 'number',
        },
    },

    providesContext: {
        'my-plugin/recordId': 'recordId',
    },
}

/**
 * @see ./edit.js
 */
edit: Edit,

/**
 * @see ./save.js
 */
save,
} );

registerBlockType( 'my-plugin/record-title', {
    title: 'Record Title',
    category: 'widgets',

    usesContext: [ 'my-plugin/recordId' ],

    edit( { context } ) {
        return 'The record ID: ' + context[ 'my-plugin/recordId' ];
    },

    save() {
        return null;
    },
} );
```

1. Edit `src/edit.js` for the `record` block. Replace `Edit` function by following code.

```
import { TextControl } from '@wordpress/components';
import { InnerBlocks } from '@wordpress/block-editor';

export default function Edit( props ) {
    const MY_TEMPLATE = [ [ 'my-plugin/record-title', {} ] ];
    const {
```

```

        attributes: { recordId },
        setAttributes,
    } = props;
    return (
        <div>
            <TextControl
                label={ __( 'Record ID:' ) }
                value={ recordId }
                onChange={ ( val ) =>
                    setAttributes( { recordId: Number( val ) } )
                }
            />
            <InnerBlocks template={ MY_TEMPLATE } templateLock="all" />
        </div>
    );
}

```

1. Edit `src/save.js` for the `record` block. Replace `save` function by following code.

```

export default function save( props ) {
    return <p>The record ID: { props.attributes.recordId }</p>;
}

```

1. Create new post and add the `record` block. If you type a number in the text box, you'll see the same number is shown in the `record-title` block below it.

The screenshot shows the WordPress editor interface. At the top, there's a toolbar with icons for adding a new block (+), text, link, and others. To the right of the toolbar are buttons for 'Save draft', 'Preview', 'Publish', settings, and more options. The main content area has a large heading 'Block Context Example'. Below the heading is a 'Record' block, which contains a text input field with the value '2001'. Underneath the block, the text 'The record ID: 2001' is displayed. In the bottom left corner of the content area, there's a small footer that says 'Document → Record'.

First published

April 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Context”](#)

[Previous Attributes](#) [Previous: Attributes](#)
[Next Deprecation](#) [Next: Deprecation](#)

Deprecation

In this article

Table of Contents

- [Example:](#)
- [Example:](#)
- [Changing the attributes set](#)
 - [Example:](#)
- [Changing the innerBlocks](#)
 - [Example:](#)

[↑ Back to top](#)

This page provides a comprehensive guide to the principles and usage of the Deprecation API. For an introduction check out the [tutorial on the basics of block deprecation](#) which can be found on the [Developer Blog](#).

When updating static blocks markup and attributes, block authors need to consider existing posts using the old versions of their block. To provide a good upgrade path, you can choose one of the following strategies:

- Do not deprecate the block and create a new one (a different name)
- Provide a “deprecated” version of the block allowing users opening these in the block editor to edit them using the updated block.

A block can have several deprecated versions. A deprecation will be tried if the current state of a parsed block is invalid, or if the deprecation defines an `isEligible` function that returns true.

Deprecations do not operate as a chain of updates in the way other software data updates, like database migrations, do. At first glance, it is easy to think that each deprecation is going to make the required changes to the data and then hand this new form of the block onto the next deprecation to make its changes. What happens instead is:

1. If the current `save` method does not produce a valid block the first deprecation in the deprecations array is passed the original saved content.

2. If its `save` method produces valid content this deprecation is used to parse the block attributes. If it has a `migrate` method it will also be run using the attributes parsed by the deprecation.
3. If the first deprecation's `save` method does not produce a valid block the subsequent deprecations in the array are tried until one producing a valid block is encountered.
4. The attributes, and any `innerBlocks`, from the first deprecation to generate a valid block are then passed back to the current `save` method to generate new valid content for the block.
5. At this point the current block should now be in a valid state and the deprecations workflow stops.

It is important to note that if a deprecation's `save` method does not produce a valid block then it is skipped completely, including its `migrate` method, even if `isEligible` would return true for the given attributes. This means that if you have several deprecations for a block and want to perform a new migration, like moving content to `InnerBlocks`, you may need to update the `migrate` methods in multiple deprecations in order for the required changes to be applied to all previous versions of the block.

It is also important to note that if a deprecation's `save` method imports additional functions from other files, changes to those files may accidentally change the behavior of the deprecation. You may want to add a snapshot copy of these functions to the deprecations file instead of importing them in order to avoid inadvertently breaking the deprecations.

For blocks with multiple deprecations, it may be easier to save each deprecation to a constant with the version of the block it applies to, and then add each of these to the block's `deprecated` array. The deprecations in the array should be in reverse chronological order. This allows the block editor to attempt to apply the most recent and likely deprecations first, avoiding unnecessary and expensive processing.

Example:

```
const v1 = {};
const v2 = {};
const v3 = {};
const deprecated = [ v3, v2, v1 ];
```

It is also recommended to keep [fixtures](#) which contain the different versions of the block content to allow you to easily test that new deprecations and migrations are working across all previous versions of the block.

Deprecations are defined on a block type as its `deprecated` property, an array of deprecation objects where each object takes the form:

- `attributes` (Object): The [attributes definition](#) of the deprecated form of the block.
- `supports` (Object): The [supports definition](#) of the deprecated form of the block.
- `save` (Function): The [save implementation](#) of the deprecated form of the block.
- `migrate`: (Function, Optional). A function which, given the old attributes and inner blocks is expected to return either the new attributes or a tuple array of attributes and inner blocks compatible with the block. As mentioned above, a deprecation's `migrate` will not be run if its `save` function does not return a valid block so you will need to make sure your migrations are available in all the deprecations where they are relevant.

- *Parameters*

- `attributes`: The block's old attributes.
- `innerBlocks`: The block's old inner blocks.

- *Return*
 - Object | Array: Either the updated block attributes or tuple array [attributes, innerBlocks].
- **isEligible**: (Function, Optional). A function which returns true if the deprecation can handle the block migration even if the block is valid. It is particularly useful in cases where a block is technically valid even once deprecated, but still requires updates to its attributes or inner blocks. This function is **not** called when the results of all previous deprecations' save functions were invalid.
 - *Parameters*
 - **attributes**: The raw block attributes as parsed from the serialized HTML, and before the block type code is applied.
 - **innerBlocks**: The block's current inner blocks.
 - **data**: An object containing properties representing the block node and its resulting block object.
 - **data.blockNode**: The raw form of the block as a result of parsing the serialized HTML.
 - **data.block**: The block object, which is the result of applying the block type to the blockNode.
 - *Return*
 - **boolean**: Whether or not this otherwise valid block is eligible to be migrated by this deprecation.

It's important to note that **attributes**, **supports**, and **save** are not automatically inherited from the current version, since they can impact parsing and serialization of a block, so they must be defined on the deprecated object in order to be processed during a migration.

Example:

```
const { registerBlockType } = wp.blocks;
const attributes = {
  text: {
    type: 'string',
    default: 'some random value',
  },
};
const supports = {
  className: false,
};

registerBlockType( 'gutenberg/block-with-deprecated-version', {
  // ... other block properties go here

  attributes,
  supports,
  save( props ) {
    return <div>{ props.attributes.text }</div>;
  },
  deprecated: [
    {
      attributes,
```

```

        supports,

        save( props ) {
            return <p>{ props.attributes.text }</p>;
        },
    ],
} );

```

In the example above we updated the markup of the block to use a `div` instead of `p`.

Changing the attributes set

Sometimes, you need to update the attributes set to rename or modify old attributes.

Example:

```

const { registerBlockType } = wp.blocks;

registerBlockType( 'gutenberg/block-with-deprecated-version', {
    // ... other block properties go here

    attributes: {
        content: {
            type: 'string',
            default: 'some random value',
        },
    },
    save( props ) {
        return <div>{ props.attributes.content }</div>;
    },
    deprecated: [
        {
            attributes: {
                text: {
                    type: 'string',
                    default: 'some random value',
                },
            },
            migrate( { text } ) {
                return {
                    content: text,
                };
            },
            save( props ) {
                return <p>{ props.attributes.text }</p>;
            },
        },
    ],
}

```

```
    ] ,  
} );
```

In the example above we updated the markup of the block to use a `div` instead of `p` and rename the `text` attribute to `content`.

Changing the innerBlocks

Situations may exist where when migrating the block we may need to add or remove innerBlocks. E.g: a block wants to migrate a title attribute to a paragraph innerBlock.

Example:

```
const { registerBlockType } = wp.blocks;  
  
registerBlockType( 'gutenberg/block-with-deprecated-version' , {  
    // ... block properties go here  
  
    save( props ) {  
        return <p>{ props.attributes.title }</p>;  
    },  
  
    deprecated: [  
        {  
            attributes: {  
                title: {  
                    type: 'string',  
                    source: 'html',  
                    selector: 'p',  
                },  
            },  
            migrate( attributes, innerBlocks ) {  
                const { title, ...restAttributes } = attributes;  
  
                return [  
                    restAttributes,  
                    [  
                        createBlock( 'core/paragraph', {  
                            content: attributes.title,  
                            fontSize: 'large',  
                        } ),  
                        ...innerBlocks,  
                    ],  
                ];  
            },  
            save( props ) {  
                return <p>{ props.attributes.title }</p>;  
            },  
        ],  
    } );
```

In the example above we updated the block to use an inner Paragraph block with a title instead of a title attribute.

Above are example cases of block deprecation. For more, real-world examples, check for deprecations in the [core block library](#). Core blocks have been updated across releases and contain simple and complex deprecations.

First published

April 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Deprecation”](#)

[Previous Context](#) [Previous: Context](#)
[Next Edit and Save](#) [Next: Edit and Save](#)

Edit and Save

In this article

Table of Contents

- [Edit](#)
 - [Block wrapper props](#)
 - [attributes](#)
 - [isSelected](#)
 - [setAttributes](#)
- [Save](#)
 - [block wrapper props](#)
 - [attributes](#)
- [Examples](#)
 - [Saving Attributes to Child Elements](#)
 - [Saving Attributes via Serialization](#)
- [Validation](#)
 - [Validation FAQ](#)

[↑ Back to top](#)

When registering a block with JavaScript on the client, the `edit` and `save` functions provide the interface for how a block is going to be rendered within the editor, how it will operate and be manipulated, and how it will be saved.

Edit

The `edit` function describes the structure of your block in the context of the editor. This represents what the editor will render when the block is used.

```
import { useBlockProps } from '@wordpress/block-editor';

// ...
const blockSettings = {
    apiVersion: 3,

    // ...

    edit: () => {
        const blockProps = useBlockProps();

        return <div { ...blockProps }>Your block.</div>;
    },
};


```

Block wrapper props

The first thing to notice here is the use of the `useBlockProps` React hook on the block wrapper element. In the example above, the block wrapper renders a “div” in the editor, but in order for the Gutenberg editor to know how to manipulate the block, add any extra classNames that are needed for the block... the block wrapper element should apply props retrieved from the `useBlockProps` react hook call. The block wrapper element should be a native DOM element, like `<div>` and `<table>`, or a React component that forwards any additional props to native DOM elements. Using a `<Fragment>` or `<ServerSideRender>` component, for instance, would be invalid.

If the element wrapper needs any extra custom HTML attributes, these need to be passed as an argument to the `useBlockProps` hook. For example to add a `my-random-classname` className to the wrapper, you can use the following code:

```
import { useBlockProps } from '@wordpress/block-editor';

// ...
const blockSettings = {
    apiVersion: 3,

    // ...

    edit: () => {
        const blockProps = useBlockProps( {
            className: 'my-random-classname',
        });

        return <div { ...blockProps }>Your block.</div>;
    },
};


```

attributes

The `edit` function also receives a number of properties through an object argument. You can use these properties to adapt the behavior of your block.

The `attributes` property surfaces all the available attributes and their corresponding values, as described by the `attributes` property when the block type was registered. See [attributes documentation](#) for how to specify attribute sources.

In this case, assuming we had defined an attribute of `content` during block registration, we would receive and use that value in our `edit` function:

```
edit: ( { attributes } ) => {
  const blockProps = useBlockProps();

  return <div { ...blockProps }>{ attributes.content }</div>;
};
```

The value of `attributes.content` will be displayed inside the `div` when inserting the block in the editor.

isSelected

The `isSelected` property is an boolean that communicates whether the block is currently selected.

```
edit: ( { attributes, isSelected } ) => {
  const blockProps = useBlockProps();

  return (
    <div { ...blockProps }>
      Your block.
      { isSelected &&
        <span>Shows only when the block is selected.</span>
      }
    </div>
  );
};
```

setAttributes

This function allows the block to update individual attributes based on user interactions.

```
edit: ( { attributes, setAttributes, isSelected } ) => {
  const blockProps = useBlockProps();

  // Simplify access to attributes
  const { content, mySetting } = attributes;

  // Toggle a setting when the user clicks the button
  const toggleSetting = () => setAttributes( { mySetting: ! mySetting } )
  return (
    <div { ...blockProps }>
      { content }
      { isSelected &&
        <span>Shows only when the block is selected.</span>
      }
    </div>
  );
};
```

```

        <button onClick={ toggleSetting }>Toggle setting</button>
    )
</div>
);
}
;
```

When using attributes that are objects or arrays it's a good idea to copy or clone the attribute prior to updating it:

```

// Good - a new array is created from the old list attribute and a new lis
const { list } = attributes;
const addListItem = ( newListItem ) =>
    setAttributes( { list: [ ...list, newListItem ] } );

// Bad - the list from the existing attribute is modified directly to add
const { list } = attributes;
const addListItem = ( newListItem ) => {
    list.push( newListItem );
    setAttributes( { list } );
};

```

Why do this? In JavaScript, arrays and objects are passed by reference, so this practice ensures changes won't affect other code that might hold references to the same data. Furthermore, the Gutenberg project follows the philosophy of the Redux library that [state should be immutable](#)—data should not be changed directly, but instead a new version of the data created containing the changes.

Save

The `save` function defines the way in which the different attributes should be combined into the final markup, which is then serialized into `post_content`.

```

save: () => {
    const blockProps = useBlockProps.save();

    return <div { ...blockProps }> Your block. </div>;
};

```

For most blocks, the return value of `save` should be an [instance of WordPress Element](#) representing how the block is to appear on the front of the site.

Note: While it is possible to return a string value from `save`, it *will be escaped*. If the string includes HTML markup, the markup will be shown on the front of the site verbatim, not as the equivalent HTML node content. If you must return raw HTML from `save`, use `wp.element.RawHTML`. As the name implies, this is prone to [cross-site scripting](#) and therefore is discouraged in favor of a WordPress Element hierarchy whenever possible.

Note: The `save` function should be a pure function that depends only on the attributes used to invoke it.

It can not have any side effect or retrieve information from another source, e.g. it is not possible to use the data module inside it `select(store).selector(...)`.

This is because if the external information changes, the block may be flagged as invalid when the post is later edited ([read more about Validation](#)).

If there is a need to have other information as part of the save, developers can consider one of these two alternatives:

- Use [dynamic blocks](#) and dynamically retrieve the required information on the server.
- Store the external value as an attribute which is dynamically updated in the block's `edit` function as changes occur.

For [dynamic blocks](#), the return value of `save` could represent a cached copy of the block's content to be shown only in case the plugin implementing the block is ever disabled.

If left unspecified, the default implementation will save no markup in post content for the dynamic block, instead deferring this to always be calculated when the block is shown on the front of the site.

[**block wrapper props**](#)

Like the `edit` function, when rendering static blocks, it's important to add the block props returned by `useBlockProps.save()` to the wrapper element of your block. This ensures that the block class name is rendered properly in addition to any HTML attribute injected by the block supports API.

[**attributes**](#)

As with `edit`, the `save` function also receives an object argument including attributes which can be inserted into the markup.

```
save: ( { attributes } ) => {
  const blockProps = useBlockProps.save();

  return <div { ...blockProps }>{ attributes.content }</div>;
};
```

When saving your block, you want to save the attributes in the same format specified by the attribute source definition. If no attribute source is specified, the attribute will be saved to the block's comment delimiter. See the [Block Attributes documentation](#) for more details.

[**Examples**](#)

Here are a couple examples of using attributes, edit, and save all together. For a full working example, see the [Introducing Attributes and Editable Fields](#) section of the Block Tutorial.

[**Saving Attributes to Child Elements**](#)

```
attributes: {
  content: {
    type: 'string',
    source: 'html',
    selector: 'div'
  }
},
edit: ( { attributes, setAttributes } ) => {
  const blockProps = useBlockProps();
```

```

    const updateFieldValue = ( val ) => {
      setAttributes( { content: val } );
    }
    return (
      <div { ...blockProps }>
        <TextControl
          label='My Text Field'
          value={ attributes.content }
          onChange={ updateFieldValue }
        />
      </div>
    );
  },
  save: ( { attributes } ) => {
    const blockProps = useBlockProps.save();

    return <div { ...blockProps }> { attributes.content } </div>;
},

```

Saving Attributes via Serialization

Ideally, the attributes saved should be included in the markup. However, there are times when this is not practical, so if no attribute source is specified the attribute is serialized and saved to the block's comment delimiter.

This example could be for a dynamic block, such as the [Latest Posts block](#), which renders the markup server-side. The save function is still required, however in this case it simply returns null since the block is not saving content from the editor.

```

attributes: {
  postsToShow: {
    type: 'number',
  }
},
edit: ( { attributes, setAttributes } ) => {
  const blockProps = useBlockProps();

  return (
    <div { ...blockProps }>
      <TextControl
        label='Number Posts to Show'
        value={ attributes.postsToShow }
        onChange={ ( val ) => {
          setAttributes( { postsToShow: parseInt( val ) } );
        }}
      />
    </div>
  );
},
save: () => {

```

```
    return null;  
}
```

Validation

When the editor loads, all blocks within post content are validated to determine their accuracy in order to protect against content loss. This is closely related to the saving implementation of a block, as a user may unintentionally remove or modify their content if the editor is unable to restore a block correctly. During editor initialization, the saved markup for each block is regenerated using the attributes that were parsed from the post's content. If the newly-generated markup does not match what was already stored in post content, the block is marked as invalid. This is because we assume that unless the user makes edits, the markup should remain identical to the saved content.

If a block is detected to be invalid, the user will be prompted to choose how to handle the invalidation:



Clicking **Attempt Block Recovery** button will attempt recovery action as much as possible.

Clicking the “3-dot” menu on the side of the block displays three options:

- **Resolve:** Open Resolve Block dialog box with two buttons:
 - **Convert to HTML:** Protects the original markup from the saved post content and convert the block from its original type to the HTML block type, enabling the user to modify the HTML markup directly.
 - **Convert to Blocks:** Protects the original markup from the saved post content and convert the block from its original type to the validated block type.
- **Convert to HTML:** Protects the original markup from the saved post content and convert the block from its original type to the HTML block type, enabling the user to modify the HTML markup directly.
- **Convert to Classic Block:** Protects the original markup from the saved post content as correct. Since the block will be converted from its original type to the Classic block type, it will no longer be possible to edit the content using controls available for the original block type.

Validation FAQ

How do blocks become invalid?

The two most common sources of block invalidations are:

1. A flaw in a block's code would result in unintended content modifications. See the question below on how to debug block invalidation as a plugin author.
2. You or an external editor changed the HTML markup of the block in such a way that it is no longer considered correct.

I'm a plugin author. What should I do to debug why my blocks are being marked as invalid?

Before starting to debug, be sure to familiarize yourself with the validation step described above documenting the process for detecting whether a block is invalid. A block is invalid if its regenerated markup does not match what is saved in post content, so often this can be caused by the attributes of a block being parsed incorrectly from the saved content.

If you're using [attribute sources](#), be sure that attributes sourced from markup are saved exactly as you expect, and in the correct type (usually a '`string`' or '`number`').

When a block is detected as invalid, a warning will be logged into your browser's developer tools console. The warning will include specific details about the exact point at which a difference in markup occurred. Be sure to look closely at any differences in the expected and actual markups to see where problems are occurring.

I've changed my block's save behavior and old content now includes invalid blocks. How can I fix this?

Refer to the guide on [Deprecated Blocks](#) to learn more about how to accommodate legacy content in intentional markup changes.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Edit and Save”](#)

[Previous Deprecation](#) [Previous: Deprecation](#)

[Next Metadata in block.json](#) [Next: Metadata in block.json](#)

Metadata in block.json

In this article

Table of Contents

- [Benefits of using the metadata file](#)
- [Block API](#)
 - [API version](#)
 - [Name](#)
 - [Title](#)
 - [Category](#)
 - [Parent](#)
 - [Ancestor](#)
 - [Allowed Blocks](#)
 - [Icon](#)
 - [Description](#)
 - [Keywords](#)
 - [Version](#)

- [Text Domain](#)
- [Attributes](#)
- [Provides Context](#)
- [Context](#)
- [Selectors](#)
- [Supports](#)
- [Block Styles](#)
- [Example](#)
- [Variations](#)
- [Block Hooks](#)
- [Editor script](#)
- [Script](#)
- [View script](#)
- [Editor style](#)
- [Style](#)
- [Render](#)
- [Assets](#)
 - [WPDefinedPath](#)
 - [WPDefinedAsset](#)
 - [Frontend enqueueing](#)
- [Internationalization](#)
 - [PHP](#)
 - [JavaScript](#)
- [Backward compatibility](#)

[↑ Back to top](#)

Starting in WordPress 5.8 release, we recommend using the `block.json` metadata file as the canonical way to register block types with both PHP (server-side) and JavaScript (client-side). Here is an example `block.json` file that would define the metadata for a plugin create a notice block.

Example:

```
{
  "$schema": "https://schemas.wp.org/trunk/block.json",
  "apiVersion": 3,
  "name": "my-plugin/notice",
  "title": "Notice",
  "category": "text",
  "parent": [ "core/group" ],
  "icon": "star",
  "description": "Shows warning, error or success notices...",
  "keywords": [ "alert", "message" ],
  "version": "1.0.3",
  "textdomain": "my-plugin",
  "attributes": {
    "message": {
      "type": "string",
      "source": "html",
      "selector": ".message"
    }
  },
}
```

```

"providesContext": {
    "my-plugin/message": "message"
},
"usesContext": [ "groupId" ],
"selectors": {
    "root": ".wp-block-my-plugin-notice"
},
"supports": {
    "align": true
},
"styles": [
    { "name": "default", "label": "Default", "isDefault": true },
    { "name": "other", "label": "Other" }
],
"example": {
    "attributes": {
        "message": "This is a notice!"
    }
},
"variations": [
    {
        "name": "example",
        "title": "Example",
        "attributes": {
            "message": "This is an example!"
        }
    }
],
"editorScript": "file:./index.js",
"script": "file:./script.js",
"viewScript": [ "file:./view.js", "example-shared-view-script" ],
"editorStyle": "file:./index.css",
"style": [ "file:./style.css", "example-shared-style" ],
"render": "file:./render.php"
}

```

Benefits of using the metadata file

The block definition allows code sharing between JavaScript, PHP, and other languages when processing block types stored as JSON, and registering blocks with the `block.json` metadata file provides multiple benefits on top of it.

From a performance perspective, when themes support lazy loading assets, blocks registered with `block.json` will have their asset enqueueing optimized out of the box. The frontend CSS and JavaScript assets listed in the `style` or `script` properties will only be enqueued when the block is present on the page, resulting in reduced page sizes.

Furthermore, because the [Block Type REST API Endpoint](#) can only list blocks registered on the server, registering blocks server-side is recommended; using the `block.json` file simplifies this registration.

The [WordPress Plugins Directory](#) can detect `block.json` files, highlight blocks included in plugins, and extract their metadata. If you wish to [submit your block\(s\) to the Block Directory](#), all

blocks contained in your plugin must have a `block.json` file for the Block Directory to recognize them.

Development is improved by using a defined schema definition file. Supported editors can provide help like tooltips, autocomplete, and schema validation. To use the schema, add the following to the top of the `block.json`.

```
"$schema": "https://schemas.wp.org/trunk/block.json"
```

Check [Registration of a block](#) to learn more about how to register a block using its metadata.

[Block API](#)

This section describes all the properties that can be added to the `block.json` file to define the behavior and metadata of block types.

[API version](#)

- Type: `number`
- Optional
- Localized: No
- Property: `apiVersion`
- Default: 1

```
{ "apiVersion": 3 }
```

The version of the Block API used by the block. The most recent version is 3 and it was introduced in WordPress 6.3.

See the [the API versions documentation](#) for more details.

[Name](#)

- Type: `string`
- Required
- Localized: No
- Property: `name`

```
{ "name": "core/heading" }
```

The name for a block is a unique string that identifies a block. Names have to be structured as `namespace/block-name`, where `namespace` is the name of your plugin or theme.

Note: A block name can only contain lowercase alphanumeric characters, dashes, and at most one forward slash to designate the plugin-unique namespace prefix. It must begin with a letter.

Note: This name is used on the comment delimiters as `<!-- wp:my-plugin/book -->`. Block types in the `core` namespace do not include a namespace when serialized.

[Title](#)

- Type: `string`
- Required
- Localized: Yes

- Property: `title`

```
{ "title": "Heading" }
```

This is the display title for your block, which can be translated with our translation functions. The title will display in the Inserter and in other areas of the editor.

Note: To keep your block titles readable and accessible in the UI, try to avoid very long titles.

[Category](#)

- Type: `string`
- Optional
- Localized: No
- Property: `category`

```
{ "category": "text" }
```

Blocks are grouped into categories to help users browse and discover them.

The core provided categories are:

- `text`
- `media`
- `design`
- `widgets`
- `theme`
- `embed`

Plugins and Themes can also register [custom block categories](#).

An implementation should expect and tolerate unknown categories, providing some reasonable fallback behavior (e.g. a “text” category).

[Parent](#)

- Type: `string[]`
- Optional
- Localized: No
- Property: `parent`

```
{ "parent": [ "my-block/product" ] }
```

Setting `parent` lets a block require that it is only available when nested within the specified blocks. For example, you might want to allow an ‘Add to Cart’ block to only be available within a ‘Product’ block.

[Ancestor](#)

- Type: `string[]`
- Optional
- Localized: No
- Property: `ancestor`
- Since: WordPress 6.0.0

```
{ "ancestor": [ "my-block/product" ] }
```

The `ancestor` property makes a block available inside the specified block types at any position of the ancestor block subtree. That allows, for example, to place a ‘Comment Content’ block inside a ‘Column’ block, as long as ‘Column’ is somewhere within a ‘Comment Template’ block. In comparison to the `parent` property blocks that specify their `ancestor` can be placed anywhere in the subtree whilst blocks with a specified `parent` need to be direct children.

[Allowed Blocks](#)

- Type: `string[]`
- Optional
- Localized: No
- Property: `allowedBlocks`
- Since: WordPress 6.5.0

```
{ "allowedBlocks": [ "my-block/product" ] }
```

The `allowedBlocks` specifies which block types can be the direct children of the block. For example, a ‘List’ block can allow only ‘List Item’ blocks as children.

[Icon](#)

- Type: `string`
- Optional
- Localized: No
- Property: `icon`

```
{ "icon": "smile" }
```

An icon property should be specified to make it easier to identify a block. These can be any of [WordPress’ Dashicons](#) (slug serving also as a fallback in non-js contexts).

Note: It’s also possible to override this property on the client-side with the source of the SVG element. In addition, this property can be defined with JavaScript as an object containing background and foreground colors. This colors will appear with the icon when they are applicable e.g.: in the inserter. Custom SVG icons are automatically wrapped in the [wp.primitives.SVG](#) component to add accessibility attributes (aria-hidden, role, and focusable).

[Description](#)

- Type: `string`
- Optional
- Localized: Yes
- Property: `description`

```
{
    "description": "Introduce new sections and organize content to help vi
}
```

This is a short description for your block, which can be translated with our translation functions. This will be shown in the block inspector.

Keywords

- Type: `string[]`
- Optional
- Localized: Yes
- Property: `keywords`
- Default: `[]`

```
{ "keywords": [ "keyword1", "keyword2" ] }
```

Sometimes a block could have aliases that help users discover it while searching. For example, an image block could also want to be discovered by photo. You can do so by providing an array of unlimited terms (which are translated).

Version

- Type: `string`
- Optional
- Localized: No
- Property: `version`
- Since: WordPress 5.8.0

```
{ "version": "1.0.3" }
```

The current version number of the block, such as 1.0 or 1.0.3. It's similar to how plugins are versioned. This field might be used with block assets to control cache invalidation, and when the block author omits it, then the installed version of WordPress is used instead.

Text Domain

- Type: `string`
- Optional
- Localized: No
- Property: `textdomain`
- Since: WordPress 5.7.0

```
{ "textdomain": "my-plugin" }
```

The [gettext](#) text domain of the plugin/block. More information can be found in the [Text Domain](#) section of the [How to Internationalize your Plugin](#) page.

Attributes

- Type: `object`
- Optional
- Localized: No
- Property: `attributes`
- Default: `{}`

```
{
  "attributes": {
    "cover": {
      "type": "string",
      "source": "attribute",
```

```

        "selector": "img",
        "attribute": "src"
    },
    "author": {
        "type": "string",
        "source": "html",
        "selector": ".book-author"
    }
}

```

Attributes provide the structured data needs of a block. They can exist in different forms when they are serialized, but they are declared together under a common interface.

See the [the attributes documentation](#) for more details.

Provides Context

- Type: `object`
- Optional
- Localized: No
- Property: `providesContext`
- Default: `{}`

Context provided for available access by descendants of blocks of this type, in the form of an object which maps a context name to one of the block's own attribute.

See [the block context documentation](#) for more details.

```
{
    "providesContext": {
        "my-plugin/recordId": "recordId"
    }
}
```

Context

- Type: `string[]`
- Optional
- Localized: No
- Property: `usesContext`
- Default: `[]`

Array of the names of context values to inherit from an ancestor provider.

See [the block context documentation](#) for more details.

```
{
    "usesContext": [ "message" ]
}
```

Selectors

- Type: `object`

- Type: `object`
- Optional
- Localized: No
- Property: `selectors`
- Default: `{}`
- Since: WordPress 6.3.0

Any custom CSS selectors, keyed by `root`, feature, or sub-feature, to be used when generating block styles for `theme.json` (global styles) stylesheets.

Providing custom selectors allows more fine grained control over which styles apply to what block elements, e.g. applying typography styles only to an inner heading while colors are still applied on the outer block wrapper etc.

See the [the selectors documentation](#) for more details.

```
{
  "selectors": {
    "root": ".my-custom-block-selector",
    "color": {
      "text": ".my-custom-block-selector p"
    },
    "typography": {
      "root": ".my-custom-block-selector > h2",
      "text-decoration": ".my-custom-block-selector > h2 span"
    }
  }
}
```

Supports

- Type: `object`
- Optional
- Localized: No
- Property: `supports`
- Default: `{}`

It contains a set of options to control features used in the editor. See the [the supports documentation](#) for more details.

Block Styles

- Type: `array`
- Optional
- Localized: Yes (`label` only)
- Property: `styles`
- Default: `[]`

```
{
  "styles": [
    { "name": "default", "label": "Default", "isDefault": true },
    { "name": "other", "label": "Other" }
  ]
}
```

Block styles can be used to provide alternative styles to block. It works by adding a class name to the block's wrapper. Using CSS, a theme developer can target the class name for the block style if it is selected.

Plugins and Themes can also register [custom block style](#) for existing blocks.

Example

- Type: `object`
- Optional
- Localized: No
- Property: `example`

```
{  
    "example": {  
        "attributes": {  
            "message": "This is a notice!"  
        }  
    }  
}
```

It provides structured example data for the block. This data is used to construct a preview for the block to be shown in the Inspector Help Panel when the user mouses over the block.

See the [Example documentation](#) for more details.

Variations

- Type: `object[]`
- Optional
- Localized: Yes (`title`, `description`, and `keywords` of each variation only)
- Property: `variations`
- Since: WordPress 5.9.0

```
{  
    "variations": [  
        {  
            "name": "example",  
            "title": "Example",  
            "attributes": {  
                "level": 2,  
                "message": "This is an example!"  
            },  
            "scope": [ "block" ],  
            "isActive": [ "level" ]  
        }  
    ]  
}
```

Block Variations is the API that allows a block to have similar versions of it, but all these versions share some common functionality. Each block variation is differentiated from the others by setting some initial attributes or inner blocks. Then at the time when a block is inserted these attributes and/or inner blocks are applied.

Note: In JavaScript you can provide a function for the `isActive` property, and a React element for the `icon`. In the `block.json` file both only support strings

See the [the variations documentation](#) for more details.

Block Hooks

- Type: `object`
- Optional
- Property: `blockHooks`
- Since: WordPress 6.4.0

```
{  
  "blockHooks": {  
    "my-plugin/banner": "after"  
  }  
}
```

Block Hooks is an API that allows a block to automatically insert itself next to all instances of a given block type, in a relative position also specified by the “hooked” block. That is, a block can opt to be inserted before or after a given block type, or as its first or last child (i.e. to be prepended or appended to the list of its child blocks, respectively). Hooked blocks will appear both on the frontend and in the editor (to allow for customization by the user).

The key is the name of the block (`string`) to hook into, and the value is the position to hook into (`string`). Take a look at the [Block Hooks documentation](#) for more info about available configurations.

Editor script

- Type: `WPDefinedAsset|WPDefinedAsset[]` ([learn more](#))
- Optional
- Localized: No
- Property: `editorScript`

```
{ "editorScript": "file:./index.js" }
```

Block type editor scripts definition. They will only be enqueued in the context of the editor.

It's possible to pass a script handle registered with the `wp_register_script` function, a path to a JavaScript file relative to the `block.json` file, or a list with a mix of both ([learn more](#)).

Note: An option to pass also an array of editor scripts exists since WordPress 6.1.0.

Script

- Type: `WPDefinedAsset|WPDefinedAsset[]` ([learn more](#))
- Optional
- Localized: No
- Property: `script`

```
{ "script": "file:./script.js" }
```

Block type frontend and editor scripts definition. They will be enqueued both in the editor and when viewing the content on the front of the site.

It's possible to pass a script handle registered with the [wp_register_script](#) function, a path to a JavaScript file relative to the `block.json` file, or a list with a mix of both ([learn more](#)).

Note: An option to pass also an array of scripts exists since WordPress 6.1.0.

[View script](#)

- Type: `WPDefinedAsset|WPDefinedAsset[]` ([learn more](#))
- Optional
- Localized: No
- Property: `viewScript`
- Since: WordPress 5.9.0

```
{ "viewScript": [ "file:./view.js", "example-shared-view-script" ] }
```

Block type frontend scripts definition. They will be enqueued only when viewing the content on the front of the site.

It's possible to pass a script handle registered with the [wp_register_script](#) function, a path to a JavaScript file relative to the `block.json` file, or a list with a mix of both ([learn more](#)).

Note: An option to pass also an array of view scripts exists since WordPress 6.1.0.

[Editor style](#)

- Type: `WPDefinedAsset|WPDefinedAsset[]` ([learn more](#))
- Optional
- Localized: No
- Property: `editorStyle`

```
{ "editorStyle": "file:./index.css" }
```

Block type editor styles definition. They will only be enqueued in the context of the editor.

It's possible to pass a style handle registered with the [wp_register_style](#) function, a path to a CSS file relative to the `block.json` file, or a list with a mix of both ([learn more](#)).

Note: An option to pass also an array of editor styles exists since WordPress 5.9.0.

[Style](#)

- Type: `WPDefinedAsset|WPDefinedAsset[]` ([learn more](#))
- Optional
- Localized: No
- Property: `style`

```
{ "style": [ "file:./style.css", "example-shared-style" ] }
```

Block type frontend and editor styles definition. They will be enqueued both in the editor and when viewing the content on the front of the site.

It's possible to pass a style handle registered with the [wp_register_style](#) function, a path to a CSS file relative to the `block.json` file, or a list with a mix of both ([learn more](#)).

Note: An option to pass also an array of styles exists since WordPress 5.9.0.

Render

- Type: WPDefinedPath ([learn more](#))
- Optional
- Localized: No
- Property: `render`
- Since: WordPress 6.1.0

```
{ "render": "file:./render.php" }
```

PHP file to use when rendering the block type on the server to show on the front end. The following variables are exposed to the file:

- `$attributes` (array): The block attributes.
- `$content` (string): The block default content.
- `$block` (WP_Block): The block instance.

An example implementation of the `render.php` file defined with `render` could look like:

```
<div <?php echo get_block_wrapper_attributes(); ?>>
    <?php echo esc_html( $attributes['label'] ); ?>
</div>
```

Note: This file loads for every instance of the block type when rendering the page HTML on the server. Accounting for that is essential when declaring functions or classes in the file. The simplest way to avoid the risk of errors is to consume that shared logic from another file.

Assets

WPDefinedPath

The `WPDefinedPath` type is a subtype of `string`, where the value represents a path to a JavaScript, CSS or PHP file relative to where `block.json` file is located. The path provided must be prefixed with `file:`. This approach is based on how npm handles [local paths](#) for packages.

Example:

In `block.json`:

```
{
    "render": "file:./render.php"
}
```

WPDefinedAsset

It extends `WPDefinedPath` for JavaScript and CSS files. An alternative to the file path would be a script or style handle name referencing an already registered asset using WordPress helpers.

Example:

In `block.json`:

```
{  
    "editorScript": "file:./index.js",  
    "script": "file:./script.js",  
    "viewScript": [ "file:./view.js", "example-shared-view-script" ],  
    "editorStyle": "file:./index.css",  
    "style": [ "file:./style.css", "example-shared-style" ]  
}
```

In the context of WordPress, when a block is registered with PHP, it will automatically register all scripts and styles that are found in the `block.json` file and use file paths rather than asset handles.

That's why, the `WPDefinedAsset` type has to offer a way to mirror also the shape of params necessary to register scripts and styles using `wp_register_script` and `wp_register_style`, and then assign these as handles associated with your block using the `script`, `style`, `editor_script`, and `editor_style` block type registration settings.

It's possible to provide an object which takes the following shape:

- `handle(string)` – the name of the script. If omitted, it will be auto-generated.
- `dependencies(string[])` – an array of registered script handles this script depends on. Default value: `[]`.
- `version(string|false|null)` – string specifying the script version number, if it has one, which is added to the URL as a query string for cache busting purposes. If the version is set to `false`, a version number is automatically added equal to current installed WordPress version. If set to `null`, no version is added. Default value: `false`.

The definition is stored inside separate PHP file which ends with `.asset.php` and is located next to the JS/CSS file listed in `block.json`. WordPress will automatically detect this file through pattern matching. This option is the preferred one as it is expected it will become an option to auto-generate those asset files with `@wordpress/scripts` package.

Example:

```
build/  
└── block.json  
└── index.js  
└── index.asset.php
```

In `block.json`:

```
{ "editorScript": "file:./index.js" }
```

In `build/index.asset.php`:

```
<?php  
return array(  
    'dependencies' => array(  
        'react',  
        'wp-blocks',  
        'wp-i18n',
```

```
),
'version'      => '3be55b05081a63d8f9d0ecb466c42cf',
);
```

Frontend enqueueing

Starting in the WordPress 5.8 release, it is possible to instruct WordPress to enqueue scripts and styles for a block type only when rendered on the frontend. It applies to the following asset fields in the `block.json` file:

- `script`
- `viewScript`
- `style`

Internationalization

WordPress string discovery system can automatically translate fields marked in this document as translatable. First, you need to set the `textdomain` property in the `block.json` file that provides block metadata.

Example:

```
{
  "title": "My block",
  "description": "My block is fantastic",
  "keywords": [ "fantastic" ],
  "textdomain": "my-plugin"
}
```

PHP

In PHP, localized properties will be automatically wrapped in `_x` function calls on the backend of WordPress when executing `register_block_type`. These translations get added as an inline script to the plugin's script handle or to the `wp-block-library` script handle in WordPress core.

The way `register_block_type` processes translatable values is roughly equivalent to the following code snippet:

```
<?php
$metadata = array(
  'title'      => _x( 'My block', 'block title', 'my-plugin' ),
  'description' => _x( 'My block is fantastic!', 'block description', 'my-plugin' ),
  'keywords'    => array( _x( 'fantastic', 'block keyword', 'my-plugin' ) );
);
```

Implementation follows the existing [get_plugin_data](#) function which parses the plugin contents to retrieve the plugin's metadata, and it applies translations dynamically.

JavaScript

In JavaScript, you can use `registerBlockType` method from `@wordpress/blocks` package and pass the `metadata` object loaded from `block.json` as the first param. All localized

properties get automatically wrapped in `_x` (from `@wordpress/i18n` package) function calls similar to how it works in PHP.

Example:

```
import { registerBlockType } from '@wordpress/blocks';
import Edit from './edit';
import metadata from './block.json';

registerBlockType( metadata, {
    edit: Edit,
    // ...other client-side settings
} );
```

Backward compatibility

The existing registration mechanism (both server side and frontend) will continue to work, it will serve as low-level implementation detail for the `block.json` based registration.

Once all details are ready, Core Blocks will be migrated iteratively and third-party blocks will see warnings appearing in the console to encourage them to refactor the block registration API used.

The following properties are going to be supported for backward compatibility reasons on the client-side only. Some of them might be replaced with alternative APIs in the future:

- `edit` – see the [Edit and Save](#) documentation for more details.
- `save` – see the [Edit and Save](#) documentation for more details.
- `transforms` – see the [Transforms](#) documentation for more details.
- `deprecated` – see the [Deprecated Blocks](#) documentation for more details.
- `merge` – undocumented as of today. Its role is to handle merging multiple blocks into one.
- `getEditWrapperProps` – undocumented as well. Its role is to inject additional props to the block edit's component wrapper.

Example:

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

registerBlockType( 'my-plugin/block-name', {
    edit: function () {
        // Edit definition goes here.
    },
    save: function () {
        // Save definition goes here.
    },
    getEditWrapperProps: function () {
        // Implementation goes here.
    },
} );
```

In the case of [dynamic blocks](#) supported by WordPress, it should be still possible to register `render_callback` property using both [`register_block_type`](#) function on the server.

First published

April 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Metadata in block.json”](#)

[Previous](#) [Edit and Save](#) [Previous: Edit and Save](#)

[Next Patterns](#) [Next: Patterns](#)

Patterns

In this article

[Table of Contents](#)

- [Block patterns](#)
 - [register_block_pattern](#)
- [Unregistering block patterns](#)
 - [unregister_block_pattern](#)
- [Block pattern categories](#)
 - [register_block_pattern_category](#)
 - [unregister_block_pattern_category](#)
- [Block patterns contextual to block types and pattern transformations](#)
- [Semantic block patterns](#)

[↑ Back to top](#)

Block Patterns are predefined block layouts available from the patterns tab of the block inserter. Once inserted into content, the blocks are ready for additional or modified content and configuration.

[Block patterns](#)

[register_block_pattern](#)

The editor comes with several core block patterns. Theme and plugin authors can register additional custom block patterns using the `register_block_pattern` helper function.

The `register_block_pattern` helper function receives two arguments.

- `title`: A machine-readable title with a naming convention of `namespace/title`.
- `properties`: An array describing properties of the pattern.

The properties available for block patterns are:

- `title` (required): A human-readable title for the pattern.
- `content` (required): Block HTML Markup for the pattern.

- **description** (optional): A visually hidden text used to describe the pattern in the inserter. A description is optional but it is strongly encouraged when the title does not fully describe what the pattern does. The description will help users discover the pattern while searching.
- **categories** (optional): An array of registered pattern categories used to group block patterns. Block patterns can be shown on multiple categories. A category must be registered separately in order to be used here.
- **keywords** (optional): An array of aliases or keywords that help users discover the pattern while searching.
- **viewportWidth** (optional): An integer specifying the intended width of the pattern to allow for a scaled preview of the pattern in the inserter.
- **blockTypes** (optional): An array of block types that the pattern is intended to be used with. Each value needs to be the declared block's name.
- **postTypes** (optional): An array of post types that the pattern is restricted to be used with. The pattern will only be available when editing one of the post types passed on the array. For all the other post types, the pattern is not available at all.
- **templateTypes** (optional): An array of template types where the pattern makes sense, for example, `404` if the pattern is for a 404 page, `single-post` if the pattern is for showing a single post.
- **inserter** (optional): By default, all patterns will appear in the inserter. To hide a pattern so that it can only be inserted programmatically, set the `inserter` to `false`.
- **source** (optional): A string that denotes the source of the pattern. For a plugin registering a pattern, pass the string `plugin`. For a theme, pass the string `theme`.

The following code sample registers a block pattern named `my-plugin/my-awesome-pattern`:

```
register_block_pattern(
    'my-plugin/my-awesome-pattern',
    array(
        'title'      => ___( 'Two buttons', 'my-plugin' ),
        'description' => __x( 'Two horizontal buttons, the left button is f
        'content'     => "<!-- wp:buttons {\"align\": \"center\"} -->\n<div
    )
);
```

Note that `register_block_pattern()` should be called from a handler attached to the `init` hook.

```
function my_plugin_register_my_patterns() {
    register_block_pattern( ... );
}

add_action( 'init', 'my_plugin_register_my_patterns' );
```

Unregistering block patterns

unregister block pattern

The `unregister_block_pattern` helper function allows a previously registered block pattern to be unregistered from a theme or plugin and receives one argument.

- **title**: The name of the block pattern to be unregistered.

The following code sample unregisters the block pattern named `my-plugin/my-awesome-pattern`:

```
 unregister_block_pattern( 'my-plugin/my-awesome-pattern' );
```

Note:

`unregister_block_pattern()` should be called from a handler attached to the `init` hook.

```
function my_plugin_unregister_my_patterns() {  
    unregister_block_pattern( ... );  
}
```

```
add_action( 'init', 'my_plugin_unregister_my_patterns' );
```

Block pattern categories

Block patterns can be grouped using categories. The block editor comes with bundled categories you can use on your custom block patterns. You can also register your own block pattern categories.

register_block_pattern_category

The `register_block_pattern_category` helper function receives two arguments.

- `title`: A machine-readable title for the block pattern category.
- `properties`: An array describing properties of the pattern category.

The properties of the pattern categories include:

- `label` (required): A human-readable label for the pattern category.

The following code sample registers the category named `hero`:

```
register_block_pattern_category(  
    'hero',  
    array( 'label' => __( 'Hero', 'my-plugin' ) )  
) ;
```

Note:

`register_block_pattern_category()` should be called from a handler attached to the `init` hook.

The category will not show under Patterns unless a pattern has been assigned to that category.

```
function my_plugin_register_my_pattern_categories() {  
    register_block_pattern_category( ... );  
}  
  
add_action( 'init', 'my_plugin_register_my_pattern_categories' );
```

unregister_block_pattern_category

The `unregister_block_pattern_category` helper function allows for a previously registered block pattern category to be unregistered from a theme or plugin and receives one argument.

- `title`: The name of the block pattern category to be unregistered.

The following code sample unregisters the category named `hero`:

```
unregister_block_pattern_category( 'hero' );
```

Note:

`unregister_block_pattern_category()` should be called from a handler attached to the `init` hook.

```
function my_plugin_unregister_my_pattern_categories() {  
    unregister_block_pattern_category( ... );  
}  
  
add_action( 'init', 'my_plugin_unregister_my_pattern_categories' );
```

Block patterns contextual to block types and pattern transformations

It is possible to attach a block pattern to one or more block types. This adds the block pattern as an available transform for that block type.

Currently, these transformations are available only to simple blocks (blocks without inner blocks). In order for a pattern to be suggested, **every selected block must be present in the block pattern**.

For instance:

```
register_block_pattern(  
    'my-plugin/powerd-by-wordpress',  
    array(  
        'title'      => ___( 'Powered by WordPress', 'my-plugin' ),  
        'blockTypes' => array( 'core/paragraph' ),  
        'content'    => '<!-- wp:paragraph {"backgroundColor":"black","textColor":"white"} -->  
        <p class="has-white-color has-black-background-color has-text-color">  
        <!-- /wp:paragraph -->',  
    )  
) ;
```

The above code registers a block pattern named `my-plugin/powerd-by-wordpress` and shows the pattern in the “transform menu” of paragraph blocks. The transformation result will keep the paragraph’s existing content and apply the other attributes – in this case, the background and text color.

As mentioned above, pattern transformations for simple blocks can also work if we have selected multiple blocks and there are matching contextual patterns to these blocks. Let’s see an example of a pattern where two block types are attached.

```

register_block_pattern(
    'my-plugin/powerd-by-wordpress',
    array(
        'title'      => __( 'Powered by WordPress', 'my-plugin' ),
        'blockTypes' => array( 'core/paragraph', 'core/heading' ),
        'content'    => '<!-- wp:group -->
                            <div class="wp-block-group">
                                <!-- wp:heading {"fontSize":"large"} -->
                                <h2 class="has-large-font-size"><span style="color: #00008B; font-size: 1.5em; font-weight: bold;"><!-- /wp:heading -->
                                <!-- wp:paragraph {"backgroundColor":"black","textColor":"white"} -->
                                <p class="has-white-color has-black-background-color">Content of my block pattern</p>
                                <!-- /wp:paragraph -->
                            </div><!-- /wp:group -->',
    )
);

```

In the above example, if we select **one of the two** block types, either a paragraph or a heading block, this pattern will be suggested by transforming the selected block using its content and will also add the remaining blocks from the pattern. If, on the other hand, we multi-select one paragraph and one heading block, both blocks will be transformed.

Blocks can also use these contextual block patterns in other places. For instance, when inserting a new Query Loop block, the user is provided with a list of all patterns attached to the block.

Semantic block patterns

In block themes, you can also mark block patterns as “header” or “footer” patterns (template part areas). We call these “semantic block patterns”. These patterns are shown to the user when inserting or replacing header or footer template parts.

Example:

```

<?php
register_block_pattern(
    'my-plugin/my-header',
    array(
        'title'      => __( 'My Header', 'my-plugin' ),
        'categories' => array( 'header' ),
        // Assigning the pattern the "header" area.
        'blockTypes' => array( 'core/template-part/header' ),
        'content'    => 'Content of my block pattern',
    )
);

```

First published

April 21, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: Patterns”](#)

[Previous Metadata in block.json](#) [Previous: Metadata in block.json](#)

[Next Registration](#) [Next: Registration](#)

Registration

In this article

Table of Contents

- [registerBlockType](#)
 - [Block Name](#)
 - [Block configuration](#)
- [Block collections](#)
- [registerBlockCollection](#)
 - [Namespace](#)
 - [Settings](#)

[↑ Back to top](#)

Block registration API reference.

You can use the functions documented on this page to register a block with JavaScript only on the client, but the recommended method is to register new block types also with PHP on the server using the ‘block.json’ metadata file. See [metadata documentation for complete information](#)

[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.

[registerBlockType](#)

- **Type:** Function

Every block starts by registering a new block type definition. To register, you use the `registerBlockType` function from the [wp-blocks package](#). The function takes two arguments, a block name and a block configuration object.

[Block Name](#)

- **Type:** String

The name for a block is a unique string that identifies a block. Names have to be structured as `namespace/block-name`, where `namespace` is the name of your plugin or theme.

```
// Registering my block with a unique name
registerBlockType( 'my-plugin/book', {} );
```

Note: A block name can only contain lowercase alphanumeric characters and dashes, and must begin with a letter.

Note: This name is used on the comment delimiters as <!-- wp:my-plugin/book -->. Those blocks provided by core don't include a namespace when serialized.

Block configuration

- **Type:** Object [{ key: value }]

A block requires a few properties to be specified before it can be registered successfully. These are defined through a configuration object, which includes the following:

title

- **Type:** String

This is the display title for your block, which can be translated with our translation functions. The title will display in the Inserter and in other areas of the editor.

```
// Our data object
title: __( 'Book' );
```

Note: To keep your block titles readable and accessible in the UI, try to avoid very long titles.

description (optional)

- **Type:** String

This is a short description for your block, which can be translated with our translation functions. This will be shown in the Block Tab in the Settings Sidebar.

```
description: __( 'Block showing a Book card.' );
```

category

- **Type:** String [text | media | design | widgets | theme | embed]

Blocks are grouped into categories to help users browse and discover them.

The core provided categories are:

- text
- media
- design
- widgets
- theme
- embed

```
// Assigning to the 'widgets' category
category: 'widgets',
```

Plugins and Themes can also register [custom block categories](#).

icon (optional)

- **Type:** String | Object

An icon property should be specified to make it easier to identify a block. These can be any of [WordPress' Dashicons](#), or a custom svg element.

```
// Specifying a dashicon for the block
icon: 'book-alt',

// Specifying a custom svg for the block
icon: <svg viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg"><path fi
```

Note: Custom SVG icons are automatically wrapped in the [wp.primitives.SVG component](#) to add accessibility attributes (aria-hidden, role, and focusable).

An object can also be passed as icon, in this case, icon, as specified above, should be included in the src property.

Besides src the object can contain background and foreground colors, this colors will appear with the icon when they are applicable e.g.: in the inserter.

```
icon: {
    // Specifying a background color to appear with the icon e.g.: in the
    background: '#7e70af',
    // Specifying a color for the icon (optional: if not set, a readable c
    foreground: '#fff',
    // Specifying an icon for the block
    src: <svg viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg"><path
} ,
```

keywords (optional)

- **Type:** Array

Sometimes a block could have aliases that help users discover it while searching. For example, an image block could also want to be discovered by photo. You can do so by providing an array of terms (which can be translated).

```
// Make it easier to discover a block with keyword aliases.
// These can be localised so your keywords work across locales.
keywords: [ __( 'image' ), __( 'photo' ), __( 'pics' ) ],
```

styles (optional)

- **Type:** Array

Block styles can be used to provide alternative styles to block. It works by adding a class name to the block's wrapper. Using CSS, a theme developer can target the class name for the block style if it is selected.

```
// Register block styles.
styles: [
    // Mark style as default.
```

```

{
  name: 'default',
  label: __( 'Rounded' ),
  isDefault: true
},
{
  name: 'outline',
  label: __( 'Outline' )
},
{
  name: 'squared',
  label: __( 'Squared' )
},
],

```

Plugins and Themes can also register [custom block style](#) for existing blocks.

attributes (optional)

- **Type:** Object

Attributes provide the structured data needs of a block. They can exist in different forms when they are serialized, but they are declared together under a common interface.

```
// Specifying my block attributes
attributes: {
  cover: {
    type: 'string',
    source: 'attribute',
    selector: 'img',
    attribute: 'src',
  },
  author: {
    type: 'string',
    source: 'html',
    selector: '.book-author',
  },
  pages: {
    type: 'number',
  },
},

```

- See: [Attributes](#).

example (optional)

- **Type:** Object

Example provides structured example data for the block. This data is used to construct a preview for the block to be shown in the Inspector Help Panel when the user mouses over the block and in the Styles panel when the block is selected.

The data provided in the example object should match the attributes defined. For example:

```
example: {
    attributes: {
        cover: 'https://example.com/image.jpg',
        author: 'William Shakespeare',
        pages: 500
    },
},
```

If `example` is not defined, the preview will not be shown. So even if no attributes are defined, setting an empty example object `example: {}` will trigger the preview to show.

It's also possible to extend the block preview with inner blocks via `innerBlocks`. For example:

```
example: {
    attributes: {
        cover: 'https://example.com/image.jpg',
    },
    innerBlocks: [
        {
            name: 'core/paragraph',
            attributes: {
                /* translators: example text. */
                content: __(
                    'Lorem ipsum dolor sit amet, consectetur adipiscing el'
                ),
            },
        },
    ],
},
```

It's also possible to define the width of the preview container in pixels via `viewportWidth`. For example:

```
example: {
    attributes: {
        cover: 'https://example.com/image.jpg',
    },
    viewportWidth: 800
},
```

variations (optional)

- **Type:** Object[]
- **Since:** WordPress 5.9.0

Similarly to how the block's styles can be declared, a block type can define block variations that the user can pick from. The difference is that, rather than changing only the visual appearance, this field provides a way to apply initial custom attributes and inner blocks at the time when a block is inserted. See the [Block Variations API](#) for more details.

supports (optional)

- **Type:** Object

Supports contains a set of options to control features used in the editor. See [the supports documentation](#) for more details.

transforms (optional)

- **Type:** Object

Transforms provide rules for what a block can be transformed from and what it can be transformed to. A block can be transformed from another block, a shortcode, a regular expression, a file, or a raw DOM node. Take a look at the [Block Transforms API](#) for more info about each available transformation.

parent (optional)

- **Type:** Array

Blocks are able to be inserted into blocks that use [InnerBlocks](#) as nested content. Sometimes it is useful to restrict a block so that it is only available as a nested block. For example, you might want to allow an ‘Add to Cart’ block to only be available within a ‘Product’ block.

Setting `parent` lets a block require that it is only available when nested within the specified blocks.

```
// Only allow this block when it is nested in a Columns block
parent: [ 'core/columns' ],
```

ancestor (optional)

- **Type:** Array
- **Since:** WordPress 6.0.0

The `ancestor` property makes a block available inside the specified block types at any position of the ancestor block subtree. That allows, for example, to place a ‘Comment Content’ block inside a ‘Column’ block, as long as ‘Column’ is somewhere within a ‘Comment Template’ block. In comparison to the `parent` property, blocks that specify their `ancestor` can be placed anywhere in the subtree whilst blocks with a specified `parent` need to be direct children.

```
// Only allow this block when it is nested at any level in a Columns block
ancestor: [ 'core/columns' ],
```

allowedBlocks (optional)

- **Type:** Array
- **Since:** WordPress 6.5.0

Setting the `allowedBlocks` property will limit which block types can be nested as direct children of the block.

```
// Only allow the Columns block to be nested as direct child of this block
allowedBlocks: [ 'core/columns' ],
```

blockHooks (optional)

- **Type:** Object
- **Since:** WordPress 6.4.0

Block Hooks is an API that allows a block to automatically insert itself next to all instances of a given block type, in a relative position also specified by the “hooked” block. That is, a block can opt to be inserted before or after a given block type, or as its first or last child (i.e. to be prepended or appended to the list of its child blocks, respectively). Hooked blocks will appear both on the frontend and in the editor (to allow for customization by the user).

The key is the name of the block (`string`) to hook into, and the value is the position to hook into (`string`). Allowed target values are:

- `before` – inject before the target block.
- `after` – inject after the target block.
- `firstChild` – inject before the first inner block of the target container block.
- `lastChild` – inject after the last inner block of the target container block.

```
{  
  blockHooks: {  
    'core/verse': 'before'  
    'core/spacer': 'after',  
    'core/column': 'firstChild',  
    'core/group': 'lastChild',  
  }  
}
```

It’s crucial to emphasize that the Block Hooks feature is only designed to work with *static* block-based templates, template parts, and patterns. For patterns, this includes those provided by the theme, from [Block Pattern Directory](#), or from calls to [register_block_pattern](#).

Block Hooks will not work with post content or patterns crafted by the user, such as synced patterns, or theme templates and template parts that have been modified by the user.

Block collections

registerBlockCollection

- **Type:** Function

Blocks can be added to collections, grouping together all blocks from the same origin

`registerBlockCollection` takes two parameters, `namespace` and an object of settings including `title` and `icon`.

Namespace

- **Type:** String

This should match the namespace declared in the block name; the name of your plugin or theme.

Settings

Title

- **Type:** String

This will display in the block inserter section, which will list all blocks in this collection.

Icon

- **Type:** Object

(Optional) An icon to display alongside the title in the block inserter.

```
// Registering a block collection
registerBlockCollection( 'my-plugin', { title: 'My Plugin' } );
```

First published

April 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Registration”](#)

[Previous Patterns](#) [Previous: Patterns](#)
[Next Selectors](#) [Next: Selectors](#)

Selectors

In this article

Table of Contents

- [Root selector](#)
 - [Example](#)
- [Feature selectors](#)
 - [Example](#)
- [Subfeature selectors](#)
 - [Example](#)
- [Shorthand](#)
- [Fallbacks](#)
 - [Example](#)

[↑ Back to top](#)

This API was stabilized in Gutenberg 15.5 and is planned for core release in WordPress 6.3. To use this prior to WordPress 6.3, you will need to install and activate Gutenberg >= 15.5.

Block Selectors is the API that allows blocks to customize the CSS selector used when their styles are generated.

A block may customize its CSS selectors at three levels: root, feature, and subfeature.

Root selector

The root selector is the block's primary CSS selector.

All blocks require a primary CSS selector for their style declarations to be included under. If one is not provided through the Block Selectors API, a default is generated in the form of `.wp-block-<name>`.

Example

```
{  
  ...  
  "selectors": {  
    "root": ".my-custom-block-selector"  
  }  
}
```

Feature selectors

Feature selectors relate to styles for a block support, e.g. border, color, typography, etc.

A block may wish to apply the styles for specific features to different elements within a block. An example might be using colors on the block's wrapper but applying the typography styles to an inner heading only.

Example

```
{  
  ...  
  "selectors": {  
    "root": ".my-custom-block-selector",  
    "color": ".my-custom-block-selector",  
    "typography": ".my-custom-block-selector > h2"  
  }  
}
```

Subfeature selectors

These selectors relate to individual styles provided by a block support e.g. `background-color`

A subfeature can have styles generated under its own unique selector. This is especially useful where one block support subfeature can't be applied to the same element as the support's other subfeatures.

A great example of this is `text-decoration`. Web browsers render this style differently, making it difficult to override if added to a wrapper element. By assigning `text-decoration` a custom selector, its style can target only the elements to which it should be applied.

Example

```
{  
  ...  
  "selectors": {  
    "root": ".my-custom-block-selector",  
    "color": ".my-custom-block-selector",  
    "typography": {  
      "root": ".my-custom-block-selector > h2",  
      "text-decoration": ".my-custom-block-selector > h2 span"  
    }  
  }  
}
```

Shorthand

Rather than specify a CSS selector for every subfeature, you can set a single selector as a string value for the relevant feature. This is the approach demonstrated for the `color` feature in the earlier examples above.

Fallbacks

A selector that hasn't been configured for a specific feature will fall back to the block's root selector. Similarly, if a subfeature hasn't had a custom selector set, it will fall back to its parent feature's selector and, if unavailable, fall back further to the block's root selector.

Rather than repeating selectors for multiple subfeatures, you can set the common selector as the parent feature's `root` selector and only define the unique selectors for the subfeatures that differ.

Example

```
{  
  ...  
  "selectors": {  
    "root": ".my-custom-block-selector",  
    "color": {  
      "text": ".my-custom-block-selector p"  
    },  
    "typography": {  
      "root": ".my-custom-block-selector > h2",  
      "text-decoration": ".my-custom-block-selector > h2 span"  
    }  
  }  
}
```

```
        }
    }
}
```

The `color.background-color` subfeature isn't explicitly set in the above example. As the `color` feature also doesn't define a `root` selector, `color.background-color` would be included under the block's primary root selector, `.my-custom-block-selector`.

For a subfeature such as `typography.font-size`, it would fallback to its parent feature's selector given that is present, i.e. `.my-custom-block-selector > h2`.

First published

April 13, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Selectors”](#)

[Previous Registration](#) [Previous: Registration](#)
[Next Styles](#) [Next: Styles](#)

Styles

In this article

Table of Contents

- [Server-side registration helper](#)
 - [register_block_style](#)
 - [unregister_block_style](#)

[↑ Back to top](#)

Block Styles allow alternative styles to be applied to existing blocks. They work by adding a `className` to the block's wrapper. This `className` can be used to provide an alternative styling for the block if the block style is selected. See the [Getting Started with JavaScript tutorial](#) for a full example.

Example:

```
wp.blocks.registerBlockStyle( 'core/quote' , {
    name: 'fancy-quote',
    label: 'Fancy Quote',
} );
```

The example above registers a block style named `fancy-quote` to the `core/quote` block. When the user selects this block style from the styles selector, an `is-style-fancy-quote` className will be added to the block's wrapper.

By adding `isDefault: true` you can mark the registered block style as the one that is recognized as active when no custom class name is provided. It also means that there will be no custom class name added to the HTML output for the style that is marked as default.

To remove a block style use `wp.blocks.unregisterBlockStyle()`.

Example:

```
wp.blocks.unregisterBlockStyle( 'core/quote', 'large' );
```

The above removes the block style named `large` from the `core/quote` block.

Important: When unregistering a block style, there can be a [race condition](#) on which code runs first: registering the style, or unregistering the style. You want your unregister code to run last. The way to do that is specify the component that is registering the style as a dependency, in this case `wp-edit-post`. Additionally, using `wp.domReady()` ensures the unregister code runs once the dom is loaded.

Enqueue your JavaScript with the following PHP code:

```
function myguten_enqueue() {
    wp_enqueue_script(
        'myguten-script',
        plugins_url( 'myguten.js', __FILE__ ),
        array( 'wp-blocks', 'wp-dom-ready', 'wp-edit-post' ),
        filemtime( plugin_dir_path( __FILE__ ) . '/myguten.js' )
    );
}
add_action( 'enqueue_block_editor_assets', 'myguten_enqueue' );
```

The JavaScript code in `myguten.js`:

```
wp.domReady( function () {
    wp.blocks.unregisterBlockStyle( 'core/quote', 'large' );
} );
```

Server-side registration helper

While the samples provided do allow full control of block styles, they do require a considerable amount of code.

To simplify the process of registering and unregistering block styles, two server-side functions are also available: `register_block_style`, and `unregister_block_style`.

register_block_style

The `register_block_style` function receives the name of the block as the first argument and an array describing properties of the style as the second argument.

The properties of the style array must include `name` and `label`:

- `name`: The identifier of the style used to compute a CSS class.
- `label`: A human-readable label for the style.

Besides the two mandatory properties, the styles properties array should also include an `inline_style` or a `style_handle` property:

- `inline_style`: Contains inline CSS code that registers the CSS class required for the style.
- `style_handle`: Contains the handle to an already registered style that should be enqueueued in places where block styles are needed.

It is also possible to set the `is_default` property to `true` to mark one of the block styles as the default one.

The following code sample registers a style for the quote block named “Blue Quote”, and provides an inline style that makes quote blocks with the “Blue Quote” style have blue color:

```
register_block_style(
    'core/quote',
    array(
        'name'          => 'blue-quote',
        'label'         => __( 'Blue Quote', 'textdomain' ),
        'inline_style'  => '.wp-block-quote.is-style-blue-quote { color: bl
    )
);
```

Alternatively, if a stylesheet was already registered which contains the CSS for the block style, it is possible to just pass the stylesheet's handle so `register_block_style` function will make sure it is enqueueued.

The following code sample provides an example of this use case:

```
wp_register_style( 'myguten-style', get_template_directory_uri() . '/custo
// ...

register_block_style(
    'core/quote',
    array(
        'name'          => 'fancy-quote',
        'label'         => __( 'Fancy Quote', 'textdomain' ),
        'style_handle'  => 'myguten-style',
    )
);
```

[unregister_block_style](#)

`unregister_block_style` allows unregistering a block style previously registered on the server using `register_block_style`.

The function's first argument is the registered name of the block, and the name of the style as the second argument.

The following code sample unregisters the style named ‘fancy-quote’ from the quote block:

```
unregister_block_style( 'core/quote', 'fancy-quote' );
```

Important: The function `unregister_block_style` only unregisters styles that were registered on the server using `register_block_style`. The function does not unregister a style registered using client-side code.

First published

May 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Styles”](#)

[Previous Selectors](#) [Previous: Selectors](#)

[Next Supports](#) [Next: Supports](#)

Supports

In this article

[Table of Contents](#)

- [anchor](#)
- [align](#)
- [alignWide](#)
- [ariaLabel](#)
- [className](#)
- [color](#)
 - [color.background](#)
 - [color._experimentalDuotone](#)
 - [color.gradients](#)
 - [color.link](#)
 - [color.text](#)
- [customClassName](#)
- [defaultStylePicker](#)
- [dimensions](#)
- [filter](#)
 - [filter.duotone](#)
- [html](#)
- [inserter](#)
- [layout](#)
 - [layout.default](#)
 - [layout.allowSwitching](#)
 - [layout.allowEditing](#)
 - [layout.allowInheriting](#)
 - [layout.allowSizingOnChildren](#)

- [layout.allowVerticalAlignment](#)
- [layout.allowJustification](#)
- [layout.allowOrientation](#)
- [layout.allowCustomContentAndWideSize](#)
- [multiple](#)
- [reusable](#)
- [lock](#)
- [position](#)
- [spacing](#)
- [typography](#)
 - [typography.fontSize](#)
 - [typography.lineHeight](#)

[↑ Back to top](#)

Block Supports is the API that allows a block to declare support for certain features.

Opting into any of these features will register additional attributes on the block and provide the UI to manipulate that attribute.

In order for the attribute to get applied to the block the generated properties get added to the wrapping element of the block. They get added to the object you get returned from the `useBlockProps` hook.

`BlockEdit` function:

```
function BlockEdit() {
  const blockProps = useBlockProps();

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

`save` function:

```
function BlockEdit() {
  const blockProps = useBlockProps.save();

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

For dynamic blocks that get rendered via a `render_callback` in PHP you can use the `get_block_wrapper_attributes()` function. It returns a string containing all the generated properties and needs to get output in the opening tag of the wrapping block element.

`render_callback` function:

```
function render_block() {
  $wrapper_attributes = get_block_wrapper_attributes();

  return sprintf(
```

```

        '<div %1$s>%2$s</div>',
        $wrapper_attributes,
        'Hello World!'
    );
}

```

anchor

- Type: boolean
- Default value: false

Anchors let you link directly to a specific block on a page. This property adds a field to define an id for the block and a button to copy the direct link. *Important: It doesn't work with dynamic blocks yet.*

```

// Declare support for anchor links.
supports: {
    anchor: true
}

```

align

- Type: boolean or array
- Default value: false

This property adds block controls which allow to change block's alignment.

```

supports: {
    // Declare support for block's alignment.
    // This adds support for all the options:
    // left, center, right, wide, and full.
    align: true
}

supports: {
    // Declare support for specific alignment options.
    align: [ 'left', 'right', 'full' ]
}

```

When the block declares support for align, the attributes definition is extended to include an align attribute with a string type. By default, no alignment is assigned. The block can apply a default alignment by specifying its own align attribute with a default e.g.:

```

attributes: {
    align: {
        type: 'string',
        default: 'right'
    }
}

```

alignWide

- Type: boolean
- Default value: true

This property allows to enable [wide alignment](#) for your theme. To disable this behavior for a single block, set this flag to false.

```
supports: {
    // Remove the support for wide alignment.
    alignWide: false
}
```

ariaLabel

- Type: boolean
- Default value: false

ARIA-labels let you define an accessible label for elements. This property allows enabling the definition of an aria-label for the block, without exposing a UI field.

```
supports: {
    // Add the support for aria label.
    ariaLabel: true
}
```

className

- Type: boolean
- Default value: true

By default, the class `.wp-block-your-block-name` is added to the root element of your saved markup. This helps having a consistent mechanism for styling blocks that themes and plugins can rely on. If, for whatever reason, a class is not desired on the markup, this functionality can be disabled.

```
supports: {
    // Remove the support for the generated className.
    className: false
}
```

color

- Type: Object
- Default value: null
- Subproperties:
 - `background`: type boolean, default value true
 - `gradients`: type boolean, default value false
 - `link`: type boolean, default value false
 - `text`: type boolean, default value true

This value signals that a block supports some of the properties related to color. When it does, the block editor will show UI controls for the user to set their values.

Note that the `background` and `text` keys have a default value of `true`, so if the `color` property is present they'll also be considered enabled:

```
supports: {
  color: {
    // This also enables text and background UI controls.
    gradients: true // Enable gradients UI control.
  }
}
```

It's possible to disable them individually:

```
supports: {
  color: { // Text UI control is enabled.
    background: false, // Disable background UI control.
    gradients: true // Enable gradients UI control.
  }
}
```

[color.background](#)

This property adds UI controls which allow the user to apply a solid background color to a block.

When color support is declared, this property is enabled by default (along with text), so simply setting `color` will enable background color.

```
supports: {
  color: true // Enables background and text
}
```

To disable background support while keeping other color supports enabled, set to `false`.

```
supports: {
  color: {
    // Disable background support. Text color support is still enabled
    background: false
  }
}
```

When the block declares support for `color.background`, the attributes definition is extended to include two new attributes: `backgroundColor` and `style`:

- `backgroundColor`: attribute of `string` type with no default assigned.

When a user chooses from the list of preset background colors, the preset slug is stored in the `backgroundColor` attribute.

Background color presets are sourced from the `editor-color-palette` [theme support](#).

The block can apply a default preset background color by specifying its own attribute with a default e.g.:

```
    attributes: {
      backgroundColor: {
        type: 'string',
        default: 'some-preset-background-slug',
      }
    }
```

- **style**: attribute of `object` type with no default assigned.

When a custom background color is selected (i.e. using the custom color picker), the custom color value is stored in the `style.color.background` attribute.

The block can apply a default custom background color by specifying its own attribute with a default e.g.:

```
    attributes: {
      style: {
        type: 'object',
        default: {
          color: {
            background: '#aabbcc',
          }
        }
      }
    }
```

[color.experimentalDuotone](#)

Note: *Deprecated since WordPress 6.3.*

This property has been replaced by [`filter.duotone`](#).

[color.gradients](#)

This property adds UI controls which allow the user to apply a gradient background to a block.

```
supports: {
  color: {
    gradients: true,
    // Default values must be disabled if you don't want to use them w
    background: false,
    text: false
  }
}
```

Gradient presets are sourced from `editor-gradient-presets` [theme support](#).

When the block declares support for `color.gradient`, the attributes definition is extended to include two new attributes: `gradient` and `style`:

- **gradient**: attribute of `string` type with no default assigned.

When a user chooses from the list of preset gradients, the preset slug is stored in the `gradient` attribute.

The block can apply a default preset gradient by specifying its own attribute with a default e.g.:

```
attributes: {
  gradient: {
    type: 'string',
    default: 'some-preset-gradient-slug',
  }
}
```

- `style`: attribute of `object` type with no default assigned.

When a custom gradient is selected (i.e. using the custom gradient picker), the custom gradient value is stored in the `style.color.gradient` attribute.

The block can apply a default custom gradient by specifying its own attribute with a default e.g.:

```
attributes: {
  style: {
    type: 'object',
    default: {
      color: {
        gradient: 'linear-gradient(135deg,rgb(170,187,204) 0%,rg
      }
    }
  }
}
```

[color.link](#)

This property adds block controls which allow the user to set link color in a block, link color is disabled by default.

```
supports: {
  color: true // Enables only background and text
}
```

To enable link color support, set to `true`.

```
supports: {
  color: {
    link: true
  }
}
```

Link color presets are sourced from the `editor-color-palette` [theme support](#).

When the block declares support for `color.link`, the attributes definition is extended to include the `style` attribute:

- `style`: attribute of `object` type with no default assigned.

When a link color is selected, the color value is stored in the `style.elements.link.color.text` attribute.

The block can apply a default link color by specifying its own attribute with a default e.g.:

```
attributes: {
  style: {
    type: 'object',
    default: {
      elements: {
        link: {
          color: {
            text: '#ff0000',
          }
        }
      }
    }
}
```

[color.text](#)

This property adds block controls which allow the user to set text color in a block.

When color support is declared, this property is enabled by default (along with background), so simply setting color will enable text color.

```
supports: {
  color: true // Enables background and text, but not link.
}
```

To disable text color support while keeping other color supports enabled, set to `false`.

```
supports: {
  color: {
    // Disable text color support.
    text: false
  }
}
```

Text color presets are sourced from the `editor-color-palette` [theme support](#).

When the block declares support for `color.text`, the attributes definition is extended to include two new attributes: `textColor` and `style`:

- `textColor`: attribute of `string` type with no default assigned.

When a user chooses from the list of preset text colors, the preset slug is stored in the `textColor` attribute.

The block can apply a default preset text color by specifying its own attribute with a default e.g.:

```
attributes: {
  textColor: {
```

```
        type: 'string',
        default: 'some-preset-text-color-slug',
    }
}
```

- **style**: attribute of `object` type with no default assigned.

When a custom text color is selected (i.e. using the custom color picker), the custom color value is stored in the `style.color.text` attribute.

The block can apply a default custom text color by specifying its own attribute with a default e.g.:

```
attributes: {
  style: {
    type: 'object',
    default: {
      color: {
        text: '#aabbcc',
      }
    }
  }
}
```

[customClassName](#)

- Type: `boolean`
- Default value: `true`

This property adds a field to define a custom className for the block's wrapper.

```
supports: {
  // Remove the support for the custom className.
  customClassName: false
}
```

[defaultStylePicker](#)

- Type: `boolean`
- Default value: `true`

When the style picker is shown, the user can set a default style for a block type based on the block's currently active style. If you prefer not to make this option available, set this property to `false`.

```
supports: {
  // Remove the Default Style picker.
  defaultStylePicker: false
}
```

dimensions

Note: Since WordPress 6.2.

- Type: Object
- Default value: null
- Subproperties:
 - minHeight: type boolean, default value false

This value signals that a block supports some of the CSS style properties related to dimensions. When it does, the block editor will show UI controls for the user to set their values if [the theme declares support](#).

```
supports: {
    dimensions: {
        aspectRatio: true // Enable aspect ratio control.
        minHeight: true // Enable min height control.
    }
}
```

When a block declares support for a specific dimensions property, its attributes definition is extended to include the `style` attribute.

- `style`: attribute of `object` type with no default assigned. This is added when `aspectRatio` or `minHeight` support is declared. It stores the custom values set by the user, e.g.:

```
attributes: {
    style: {
        dimensions: {
            aspectRatio: "16/9",
            minHeight: "50vh"
        }
    }
}
```

filter

- Type: Object
- Default value: null
- Subproperties:
 - duotone: type boolean, default value false

This value signals that a block supports some of the properties related to filters. When it does, the block editor will show UI controls for the user to set their values.

filter.duotone

This property adds UI controls which allow the user to apply a duotone filter to a block or part of a block.

```
supports: {
    filter: {
```

```

        // Enable duotone support
        duotone: true
    }
},
selectors: {
    filter: {
        // Apply the filter to img elements inside the image block
        duotone: '.wp-block-image img'
    }
}

```

The filter can be applied to an element inside the block by setting the `selectors.filter.duotone` selector.

Duotone presets are sourced from `color.duotone` in [theme.json](#).

When the block declares support for `filter.duotone`, the attributes definition is extended to include the attribute `style`:

- `style`: attribute of `object` type with no default assigned.

The block can apply a default duotone color by specifying its own attribute with a default e.g.:

```

attributes: {
    style: {
        type: 'object',
        default: {
            color: {
                duotone: [
                    '#FFF',
                    '#000'
                ]
            }
        }
    }
}

```

html

- Type: `boolean`
- Default value: `true`

By default, a block's markup can be edited individually. To disable this behavior, set `html` to `false`.

```

supports: {
    // Remove support for an HTML mode.
    html: false
}

```

inserter

- Type: boolean
- Default value: true

By default, all blocks will appear in the inserter, block transforms menu, Style Book, etc. To hide a block from all parts of the user interface so that it can only be inserted programmatically, set `inserter` to false.

```
supports: {
    // Hide this block from the inserter.
    inserter: false
}
```

layout

- Type: boolean or Object
- Default value: null
- Subproperties:
 - `default`: type Object, default value null
 - `allowSwitching`: type boolean, default value false
 - `allowEditing`: type boolean, default value true
 - `allowInheriting`: type boolean, default value true
 - `allowSizingOnChildren`: type boolean, default value false
 - `allowVerticalAlignment`: type boolean, default value true
 - `allowJustification`: type boolean, default value true
 - `allowOrientation`: type boolean, default value true
 - `allowCustomContentAndWideSize`: type boolean, default value true

This value only applies to blocks that are containers for inner blocks. If set to true the layout type will be `flow`. For other layout types it's necessary to set the `type` explicitly inside the `default` object.

layout.default

- Type: Object
- Default value: null

Allows setting the `type` property to define what layout type is default for the block, and also default values for any properties inherent to that layout type, e.g., for a `flex` layout, a default value can be set for `flexWrap`.

layout.allowSwitching

- Type: boolean
- Default value: false

Exposes a switcher control that allows toggling between all existing layout types.

layout.allowEditing

- Type: boolean

- Default value: `true`

Determines display of layout controls in the block sidebar. If set to false, layout controls will be hidden.

[layout.allowInheriting](#)

- Type: `boolean`
- Default value: `true`

For the `flow` layout type only, determines display of the “Inner blocks use content width” toggle.

[layout.allowSizingOnChildren](#)

- Type: `boolean`
- Default value: `false`

For the `flex` layout type only, determines display of sizing controls (Fit/Fill/Fixed) on all child blocks of the flex block.

[layout.allowVerticalAlignment](#)

- Type: `boolean`
- Default value: `true`

For the `flex` layout type only, determines display of the vertical alignment control in the block toolbar.

[layout.allowJustification](#)

- Type: `boolean`
- Default value: `true`

For the `flex` layout type, determines display of the justification control in the block toolbar and block sidebar. For the `constrained` layout type, determines display of justification control in the block sidebar.

[layout.allowOrientation](#)

- Type: `boolean`
- Default value: `true`

For the `flex` layout type only, determines display of the orientation control in the block toolbar.

[layout.allowCustomContentAndWideSize](#)

- Type: `boolean`
- Default value: `true`

For the `constrained` layout type only, determines display of the custom content and wide size controls in the block sidebar.

multiple

- Type: boolean
- Default value: true

A non-multiple block can be inserted into each post, one time only. For example, the built-in ‘More’ block cannot be inserted again if it already exists in the post being edited. A non-multiple block’s icon is automatically dimmed (unclickable) to prevent multiple instances.

```
supports: {
    // Use the block just once per post
    multiple: false
}
```

reusable

- Type: boolean
- Default value: true

A block may want to disable the ability of being converted into a reusable block. By default all blocks can be converted to a reusable block. If supports reusable is set to false, the option to convert the block into a reusable block will not appear.

```
supports: {
    // Don't allow the block to be converted into a reusable block.
    reusable: false,
}
```

lock

- Type: boolean
- Default value: true

A block may want to disable the ability to toggle the lock state. It can be locked/unlocked by a user from the block “Options” dropdown by default. To disable this behavior, set lock to false.

```
supports: {
    // Remove support for locking UI.
    lock: false
}
```

position

Note: Since WordPress 6.2.

- Type: Object
- Default value: null
- Subproperties:
 - sticky: type boolean, default value false

This value signals that a block supports some of the CSS style properties related to position. When it does, the block editor will show UI controls for the user to set their values if [the theme declares support](#).

Note that sticky position controls are currently only available for blocks set at the root level of the document. Setting a block to the `sticky` position will stick the block to its most immediate parent when the user scrolls the page.

```
supports: {
  position: {
    sticky: true // Enable selecting sticky position.
  }
}
```

When the block declares support for a specific position property, its attributes definition is extended to include the `style` attribute.

- `style`: attribute of `object` type with no default assigned. This is added when `sticky` support is declared. It stores the custom values set by the user, e.g.:

```
attributes: {
  style: {
    position: {
      type: "sticky",
      top: "0px"
    }
  }
}
```

[spacing](#)

- Type: `Object`
- Default value: `null`
- Subproperties:
 - `margin`: type `boolean` or `array`, default value `false`
 - `padding`: type `boolean` or `array`, default value `false`
 - `blockGap`: type `boolean` or `array`, default value `false`

This value signals that a block supports some of the CSS style properties related to spacing. When it does, the block editor will show UI controls for the user to set their values if [the theme declares support](#).

```
supports: {
  spacing: {
    margin: true, // Enable margin UI control.
    padding: true, // Enable padding UI control.
    blockGap: true, // Enables block spacing UI control for blocks that
  }
}
```

When the block declares support for a specific spacing property, its attributes definition is extended to include the `style` attribute.

- `style`: attribute of `object` type with no default assigned. This is added when `margin` or `padding` support is declared. It stores the custom values set by the user, e.g.:

```
attributes: {
  style: {
    margin: 'value',
    padding: {
      top: 'value',
    }
  }
}
```

A spacing property may define an array of allowable sides – ‘top’, ‘right’, ‘bottom’, ‘left’ – that can be configured. When such arbitrary sides are defined, only UI controls for those sides are displayed.

Axial sides are defined with the `vertical` and `horizontal` terms, and display a single UI control for each axial pair (for example, `vertical` controls both the top and bottom sides). A spacing property may support arbitrary individual sides **or** axial sides, but not a mix of both.

Note: `blockGap` accepts `vertical` and `horizontal` axial sides, which adjust gap column and row values. `blockGap` doesn't support arbitrary sides.

```
supports: {
  spacing: {
    margin: [ 'top', 'bottom' ],           // Enable margin for arbitrary sides
    padding: true,                      // Enable padding for all sides
    blockGap: [ 'horizontal', 'vertical' ], // Enables axial (column/row) gap
  }
}
```

typography

- Type: `Object`
- Default value: `null`
- Subproperties:
 - `fontSize`: type `boolean`, default value `false`
 - `lineHeight`: type `boolean`, default value `false`

The presence of this object signals that a block supports some typography related properties. When it does, the block editor will show a typography UI allowing the user to control their values.

```
supports: {
  typography: {
    // Enable support and UI control for font-size.
    fontSize: true,
    // Enable support and UI control for line-height.
    lineHeight: true,
  },
}
```

[typography.fontSize](#)

- Type: `boolean`
- Default value: `false`

This value signals that a block supports the font-size CSS style property. When it does, the block editor will show an UI control for the user to set its value.

The values shown in this control are the ones declared by the theme via the `editor-font-sizes` [theme support](#), or the default ones if none are provided.

```
supports: {
  typography: {
    // Enable support and UI control for font-size.
    fontSize: true,
  },
}
```

When the block declares support for `fontSize`, the attributes definition is extended to include two new attributes: `fontSize` and `style`:

- `fontSize`: attribute of `string` type with no default assigned. It stores any preset value selected by the user. The block can apply a default `fontSize` by specifying its own `fontSize` attribute with a default e.g.:

```
attributes: {
  fontSize: {
    type: 'string',
    default: 'some-value',
  }
}
```

- `style`: attribute of `object` type with no default assigned. It stores the custom values set by the user and is shared with other block supports such as `color`. The block can apply a default `style` by specifying its own `style` attribute with a default e.g.:

```
attributes: {
  style: {
    type: 'object',
    default: {
      typography: {
        fontSize: 'value'
      }
    }
  }
}
```

[typography.lineHeight](#)

- Type: `boolean`
- Default value: `false`

This value signals that a block supports the line-height CSS style property. When it does, the block editor will show an UI control for the user to set its value if [the theme declares support](#).

```
supports: {
    typography: {
        // Enable support and UI control for line-height.
        lineHeight: true,
    },
}
```

When the block declares support for `lineHeight`, the attributes definition is extended to include a new attribute `style` of `object` type with no default assigned. It stores the custom value set by the user. The block can apply a default style by specifying its own `style` attribute with a default e.g.:

```
attributes: {
    style: {
        type: 'object',
        default: {
            typography: {
                lineHeight: 'value'
            }
        }
    }
}
```

First published

April 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Supports”](#)

[Previous Styles](#) [Previous: Styles](#)
[Next Templates](#) [Next: Templates](#)

Templates

In this article

Table of Contents

- [API](#)
- [Block attributes](#)
- [Custom post types](#)
- [Locking](#)
- [Individual block locking](#)
- [Nested templates](#)

[↑ Back to top](#)

A block template is defined as a list of block items. Such blocks can have predefined attributes, placeholder content, and be static or dynamic. Block templates allow specifying a default initial state for an editor session.

The scope of templates include:

- Setting a default state dynamically on the client. (like `defaultBlock`)
- Registered as a default for a given post type.

Planned additions:

- Saved and assigned to pages as “page templates”.
- Defined in a `template.php` file or pulled from a custom post type (`wp_templates`) that is site specific.
- As the equivalent of the theme hierarchy.

API

Templates can be declared in JS or in PHP as an array of blockTypes (block name and optional attributes).

The first example in PHP creates a template for posts that includes an image block to start, you can add as many or as few blocks to your template as needed.

PHP example:

```
<?php
function myplugin_register_template() {
    $post_type_object = get_post_type_object( 'post' );
    $post_type_object->template = array(
        array( 'core/image' ),
    );
}
add_action( 'init', 'myplugin_register_template' );
```

The following example in JavaScript creates a new block using [InnerBlocks](#) and templates, when inserted creates a set of blocks based off the template.

```
const el = React.createElement;
const { registerBlockType } = wp.blocks;
const { InnerBlocks } = wp.blockEditor;

const BLOCKS_TEMPLATE = [
    [ 'core/image', {} ],
    [ 'core/paragraph', { placeholder: 'Image Details' } ],
];

registerBlockType( 'myplugin/template', {
    title: 'My Template Block',
    category: 'widgets',
    edit: ( props ) => {
        return el( InnerBlocks, {
            template: BLOCKS_TEMPLATE,
            templateLock: false,
        });
    }
});
```

```

        } );
    },
    save: ( props ) => {
        return el( InnerBlocks.Content, {} );
    },
} );

```

See the [Meta Block Tutorial](#) for a full example of a template in use.

Block attributes

To find a comprehensive list of all block attributes that you can define in a template, consult the block's `block.json` file, and look at the `attributes` and `supports` values.

For example, [packages/block-library/src/heading/block.json](#) shows that the block has a `level` attribute, and supports the `anchor` parameter.

If you don't have the Gutenberg plugin installed, you can find `block.json` files inside `wp-includes/blocks/heading/block.json`.

Custom post types

A custom post type can register its own template during registration:

```

function myplugin_register_book_post_type() {
    $args = array(
        'public' => true,
        'label' => 'Books',
        'show_in_rest' => true,
        'template' => array(
            array( 'core/image', array(
                'align' => 'left',
            ) ),
            array( 'core/heading', array(
                'placeholder' => 'Add Author...',
            ) ),
            array( 'core/paragraph', array(
                'placeholder' => 'Add Description...',
            ) ),
        ),
    );
    register_post_type( 'book', $args );
}
add_action( 'init', 'myplugin_register_book_post_type' );

```

Locking

Sometimes the intention might be to lock the template on the UI so that the blocks presented cannot be manipulated. This is achieved with a `template_lock` property.

```

function myplugin_register_template() {
    $post_type_object = get_post_type_object( 'post' );

```

```

$post_type_object->template = array(
    array( 'core/paragraph', array(
        'placeholder' => 'Add Description...',
    ) ),
);
$post_type_object->template_lock = 'all';
}
add_action( 'init', 'myplugin_register_template' );

```

Options:

- `contentOnly` — prevents all operations. Additionally, the block types that don't have content are hidden from the list view and can't gain focus within the block list. Unlike the other lock types, this is not overridable by children.
- `all` — prevents all operations. It is not possible to insert new blocks, move existing blocks, or delete blocks.
- `insert` — prevents inserting or removing blocks, but allows moving existing blocks.

Lock settings can be inherited by InnerBlocks. If `templateLock` is not set in an InnerBlocks area, the locking of the parent InnerBlocks area is used. If the block is a top level block, the locking configuration of the current post type is used.

Individual block locking

Alongside template level locking, you can lock individual blocks; you can do this using a `lock` attribute on the attributes level. Block-level lock takes priority over the `templateLock` feature. Currently, you can lock moving and removing blocks.

```

attributes: {
    // Prevent a block from being moved or removed.
    lock: {
        remove: true,
        move: true,
    }
}

```

Options:

- `remove` — Locks the ability of a block from being removed.
- `move` — Locks the ability of a block from being moved.

You can use this with `templateLock` to lock all blocks except a single block by using `false` in `remove` or `move`.

```

$template = array(
    array( 'core/image', array(
        'align' => 'left',
    ) ),
    array( 'core/heading', array(
        'placeholder' => 'Add Author...',
    ) ),
    // Allow a Paragraph block to be moved or removed.
    array( 'core/paragraph', array(
        'placeholder' => 'Add Description...',
```

```
        'lock' => array(
            'move'    => false,
            'remove'  => false,
        ),
    ) ),
);
)
```

Nested templates

Container blocks like the columns blocks also support templates. This is achieved by assigning a nested template to the block.

```
$template = array(
    array( 'core/paragraph', array(
        'placeholder' => 'Add a root-level paragraph',
    ) ),
    array( 'core/columns', array(), array(
        array( 'core/column', array(), array(
            array( 'core/image', array() ),
        ) ),
        array( 'core/column', array(), array(
            array( 'core/paragraph', array(
                'placeholder' => 'Add a inner paragraph'
            ) ),
        ) ),
    ) )
);
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Templates”](#)

[Previous Supports Previous: Supports](#)
[Next Transforms Next: Transforms](#)

Transforms

In this article

Table of Contents

- [Transform direction: to and from](#)
- [Transformations types](#)
 - [Block](#)

- [Enter](#)
- [Files](#)
- [Prefix](#)
- [Raw](#)
- [Shortcode](#)
- [ungroup blocks](#)

[↑ Back to top](#)

Block Transforms is the API that allows a block to be transformed *from* and *to* other blocks, as well as *from* other entities. Existing entities that work with this API include shortcodes, files, regular expressions, and raw DOM nodes.

Transform direction: to and from

A block declares which transformations it supports via the optional `transforms` key of the block configuration, whose subkeys `to` and `from` hold an array of available transforms for every direction. Example:

```
export const settings = {
  title: 'My Block Title',
  description: 'My block description',
  /* ... */
  transforms: {
    from: [
      /* supported from transforms */
    ],
    to: [
      /* supported to transforms */
    ],
  },
};
```

Transformations types

This section goes through the existing types of transformations blocks support:

- `block`
- `enter`
- `files`
- `prefix`
- `raw`
- `shortcode`

Block

This type of transformations support both *from* and *to* directions, allowing blocks to be converted into a different one. It has a corresponding UI control within the block toolbar.

A transformation of type `block` is an object that takes the following parameters:

- **type** (*string*): the value `block`.

- **blocks** (*array*): a list of known block types. It also accepts the wildcard value ("*"), meaning that the transform is available to *all* block types (eg: all blocks can transform into core/group).
- **transform** (*function*): a callback that receives the attributes and inner blocks of the block being processed. It should return a block object or an array of block objects.
- **isMatch** (*function, optional*): a callback that receives the block attributes as the first argument and the block object as the second argument and should return a boolean. Returning `false` from this function will prevent the transform from being available and displayed as an option to the user.
- **isMultiBlock** (*boolean, optional*): whether the transformation can be applied when multiple blocks are selected. If true, the `transform` function's first parameter will be an array containing each selected block's attributes, and the second an array of each selected block's inner blocks. False by default.
- **priority** (*number, optional*): controls the priority with which a transformation is applied, where a lower value will take precedence over higher values. This behaves much like a [WordPress hook](#). Like hooks, the default priority is 10 when not otherwise set.

Example: from Paragraph block to Heading block

To declare this transformation we add the following code into the heading block configuration, which uses the `createBlock` function from the [wp-blocks package](#).

```
transforms: {
  from: [
    {
      type: 'block',
      blocks: [ 'core/paragraph' ],
      transform: ( { content } ) => {
        return createBlock( 'core/heading', {
          content,
        } );
      },
    },
  ],
},
```

Example: blocks that have InnerBlocks

A block with InnerBlocks can also be transformed from and to another block with InnerBlocks.

```
transforms: {
  to: [
    {
      type: 'block',
      blocks: [ 'some/block-with-innerblocks' ],
      transform: ( attributes, innerBlocks ) => {
        return createBlock(
          'some/other-block-with-innerblocks',
          attributes,
          innerBlocks
        );
      },
    },
  ],
},
```

```
] ,  
} ,
```

Enter

This type of transformations support the *from* direction, allowing blocks to be created from some content introduced by the user. They're applied in a new block line after the user has introduced some content and hit the ENTER key.

A transformation of type `enter` is an object that takes the following parameters:

- **type** (*string*): the value `enter`.
- **regExp** (*RegExp*): the Regular Expression to use as a matcher. If the value matches, the transformation will be applied.
- **transform** (*function*): a callback that receives an object with a `content` field containing the value that has been entered. It should return a block object or an array of block objects.
- **priority** (*number, optional*): controls the priority with which a transform is applied, where a lower value will take precedence over higher values. This behaves much like a [WordPress hook](#). Like hooks, the default priority is `10` when not otherwise set.

Example: from — to Separator block

To create a separator block when the user types the hyphen three times and then hits the ENTER key we can use the following code:

```
transforms = {  
  from: [  
    {  
      type: 'enter',  
      regExp: /^-{3,}\$/,  
      transform: () => createBlock( 'core/separator' ),  
    },  
  ],  
};
```

Files

This type of transformations support the *from* direction, allowing blocks to be created from files dropped into the editor.

A transformation of type `files` is an object that takes the following parameters:

- **type** (*string*): the value `files`.
- **transform** (*function*): a callback that receives the array of files being processed. It should return a block object or an array of block objects.
- **isMatch** (*function, optional*): a callback that receives the array of files being processed and should return a boolean. Returning `false` from this function will prevent the transform from being applied.
- **priority** (*number, optional*): controls the priority with which a transform is applied, where a lower value will take precedence over higher values. This behaves much like a [WordPress hook](#). Like hooks, the default priority is `10` when not otherwise set.

Example: from file to File block

To create a File block when the user drops a file into the editor we can use the following code:

```
transforms: {
  from: [
    {
      type: 'files',
      isMatch: ( files ) => files.length === 1,
      // By defining a lower priority than the default of 10,
      // we make that the File block to be created as a fallback,
      // if no other transform is found.
      priority: 15,
      transform: ( files ) => {
        const file = files[ 0 ];
        const blobURL = createBlobURL( file );
        // File will be uploaded in componentDidMount()
        return createBlock( 'core/file', {
          href: blobURL,
          fileName: file.name,
          textLinkHref: blobURL,
        } );
      },
    },
  ],
}
```

Prefix

This type of transformations support the *from* direction, allowing blocks to be created from some text typed by the user. They're applied when, in a new block line, the user types some text and then adds a trailing space.

A transformation of type `prefix` is an object that takes the following parameters:

- **type (string)**: the value `prefix`.
- **prefix (string)**: the character or sequence of characters that match this transform.
- **transform (function)**: a callback that receives the content introduced. It should return a block object or an array of block objects.
- **priority (number, optional)**: controls the priority with which a transform is applied, where a lower value will take precedence over higher values. This behaves much like a [WordPress hook](#). Like hooks, the default priority is 10 when not otherwise set.

Example: from text to custom block

If we want to create a custom block when the user types the question mark, we could use this code:

```
transforms: {
  from: [
    {
      type: 'prefix',
      prefix: '?',
      transform( content ) {
        return createBlock( 'my-plugin/question', {
          content,
        });
      }
    },
  ],
}
```

```

        } );
    },
];
}

```

Raw

This type of transformations support the *from* direction, allowing blocks to be created from raw HTML nodes. They're applied when the user executes the “Convert to Blocks” action from within the block setting UI menu, as well as when some content is pasted or dropped into the editor.

A transformation of type `raw` is an object that takes the following parameters:

- **type** (*string*): the value `raw`.
- **transform** (*function, optional*): a callback that receives the node being processed. It should return a block object or an array of block objects.
- **schema** (*object|function, optional*): defines an [HTML content model](#) used to detect and process pasted contents. See [below](#).
- **selector** (*string, optional*): a CSS selector string to determine whether the element matches according to the [element.matches](#) method. The transform won't be executed if the element doesn't match. This is a shorthand and alternative to using `isMatch`, which, if present, will take precedence.
- **isMatch** (*function, optional*): a callback that receives the node being processed and should return a boolean. Returning `false` from this function will prevent the transform from being applied.
- **priority** (*number, optional*): controls the priority with which a transform is applied, where a lower value will take precedence over higher values. This behaves much like a [WordPress hook](#). Like hooks, the default priority is `10` when not otherwise set.

Example: from URLs to Embed block

If we want to create an Embed block when the user pastes some URL in the editor, we could use this code:

```

transforms: {
  from: [
    {
      type: 'raw',
      isMatch: ( node ) =>
        node.nodeName === 'P' &&
        /^\\s*(https?:\\/\\/\\S+)\\s*$/i.test( node.textContent ),
      transform: ( node ) => {
        return createBlock( 'core/embed', {
          url: node.textContent.trim(),
        } );
      },
    },
  ],
}

```

Schemas and Content Models

When pasting content it's possible to define a [content model](#) that will be used to validate and process pasted content. It's often the case that HTML pasted into the editor will contain a mixture of elements that *should* transfer as well as elements that *shouldn't*. For example, consider pasting `12:04 pm` into the editor. We want to copy 12:04 pm and omit the `` and its `class` attribute because those won't carry the same meaning or structure as they originally did from where they were copied.

When writing `raw` transforms you can control this by supplying a `schema` which describes allowable content and which will be applied to clean up the pasted content before attempting to match with your block. The schemas are passed into [cleanNodeList from @wordpress/dom](#); check there for a [complete description of the schema](#).

```
schema = { span: { children: { '#text': {} } } };
```

Example: a custom content model

Suppose we want to match the following HTML snippet and turn it into some kind of custom post preview block.

```
<div data-post-id="13">
  <h2>The Post Title</h2>
  <p>Some <em>great</em> content.</p>
</div>
```

We want to tell the editor to allow the inner `h2` and `p` elements. We do this by supplying the following schema. In this example we're using the function form, which accepts an argument supplying `phrasingContentSchema` (as well as a boolean `isPaste` indicating if the transformation operation started with pasting text). The `phrasingContentSchema` is pre-defined to match HTML phrasing elements, such as `` and `<sup>` and `<kbd>`.

Anywhere we expect

a `<RichText />` component is a good place to allow phrasing content otherwise we'll lose all text formatting on conversion.

```
schema = ({ phrasingContentSchema }) => {
  div: {
    required: true,
    attributes: [ 'data-post-id' ],
    children: {
      h2: { children: phrasingContentSchema },
      p: { children: phrasingContentSchema }
    }
  }
}
```

When we successfully match this content every HTML attribute will be stripped away except for `data-post-id` and if we have other arrangements of HTML inside of a given `div` then it won't match our transformer. Likewise we'd fail to match if we found an `<h3>` in there instead of an `<h2>`.

Schemas are most-important when wanting to match HTML snippets containing non-phrasing content, such as `<details>` with a `<summary>`. Without declaring the custom schema the editor will skip over these other constructions before attempting to run them through any block transforms.

Shortcode

This type of transformations support the *from* direction, allowing blocks to be created from shortcodes. It's applied as part of the `raw` transformation process.

A transformation of type `shortcode` is an object that takes the following parameters:

- **type** (*string*): the value `shortcode`.
- **tag** (*string|array*): the shortcode tag or list of shortcode aliases this transform can work with.
- **transform** (*function, optional*): a callback that receives the shortcode attributes as the first argument and the [WP_shortcodeMatch](#) as the second. It should return a block object or an array of block objects. When this parameter is defined, it will take precedence over the `attributes` parameter.
- **attributes** (*object, optional*): object representing where the block attributes should be sourced from, according to the attributes shape defined by the [block configuration object](#). If a particular attribute contains a `shortcode` key, it should be a function that receives the shortcode attributes as the first arguments and the [WP_shortcodeMatch](#) as second, and returns a value for the attribute that will be sourced in the block's comment.
- **isMatch** (*function, optional*): a callback that receives the shortcode attributes per the [Shortcode API](#) and should return a boolean. Returning `false` from this function will prevent the shortcode to be transformed into this block.
- **priority** (*number, optional*): controls the priority with which a transform is applied, where a lower value will take precedence over higher values. This behaves much like a [WordPress hook](#). Like hooks, the default priority is `10` when not otherwise set.

Example: from shortcode to block using transform

An existing shortcode can be transformed into its block counterpart using the `transform` method.

```
transforms: {
  from: [
    {
      type: 'shortcode',
      tag: 'video',
      transform( { named: { src } } ) {
        return createBlock( 'core/video', { src } );
      },
      // Prevent the shortcode to be converted
      // into this block when it doesn't
      // have the proper ID.
      isMatch( { named: { id } } ) {
        return id === 'my-id';
      },
    },
  ],
},
```

Example: from shortcode to block using attributes

An existing shortcode can be transformed into its block counterpart using the `attributes` parameters.

```

transforms: {
  from: [
    {
      type: 'shortcode',
      tag: 'youtube',
      attributes: {
        url: {
          type: 'string',
          source: 'attribute',
          attribute: 'src',
          selector: 'img',
        },
        align: {
          type: 'string',
          // The shortcode function will extract
          // the shortcode attrs into a value
          // to be sourced in the block's comment.
          shortcode: ( { named: { align = 'alignnone' } } ) => {
            return align.replace( 'align', '' );
          },
        },
      },
      // Prevent the shortcode to be converted
      // into this block when it doesn't
      // have the proper ID.
      isMatch( { named: { id } } ) {
        return id === 'my-id';
      },
    },
  ],
},

```

[ungroup blocks](#)

Via the optional `transforms` key of the block configuration, blocks can use the `ungroup` subkey to define the blocks that will replace the block being processed. These new blocks will usually be a subset of the existing inner blocks, but could also include new blocks.

If a block has an `ungroup` transform, it is eligible for ungrouping, without the requirement of being the default grouping block. The UI used to ungroup a block with this API is the same as the one used for the default grouping block. In order for the Ungroup button to be displayed, we must have a single grouping block selected, which also contains some inner blocks.

ungroup is a callback function that receives the attributes and inner blocks of the block being processed. It should return an array of block objects.

Example:

```

export const settings = {
  title: 'My grouping Block Title',
  description: 'My grouping block description',
  /* ... */
  transforms: {

```

```
        ungroup: ( attributes, innerBlocks ) =>
            innerBlocks.flatMap( ( innerBlock ) => innerBlock.innerBlocks
        } ,
    };
```

First published

April 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Transforms”](#)

[Previous Templates](#) [Previous: Templates](#)
[Next Variations](#) [Next: Variations](#)

Variations

In this article

Table of Contents

- [Defining a block variation](#)
- [Creating a block variation](#)
- [Removing a block variation](#)
- [Block variations versus block styles](#)
- [Using isDefault](#)
 - [Caveats to using isDefault](#)
- [Using isActive](#)
 - [Caveats to using isActive](#)

[↑ Back to top](#)

The Block Variations API allows you to define multiple versions (variations) of a block. A block variation differs from the original block by a set of initial attributes or inner blocks. When you insert the block variation into the Editor, these attributes and/or inner blocks are applied.

Variations are an excellent way to create iterations of existing blocks without building entirely new blocks from scratch.

To better understand this API, consider the Embed block. This block contains numerous variations for each type of embeddable content (WordPress, YouTube, etc.). Each Embed block variation shares the same underlying functionality for editing, saving, and so on. Besides the name and descriptive information, the main difference is the `providerNameSlug` attribute. Below is a simplified example of the variations in the Embed block. View the [source code](#) for the complete specification.

```

variations: [
  {
    name: 'wordpress',
    title: 'WordPress',
    description: __( 'Embed a WordPress post.' ),
    attributes: { providerNameSlug: 'wordpress' },
  },
  {
    name: 'youtube',
    title: 'YouTube',
    description: __( 'Embed a YouTube video.' ),
    attributes: { providerNameSlug: 'youtube' },
  },
],

```

Defining a block variation

A block variation is defined by an object that can contain the following fields:

- **name** (type `string`) – A unique and machine-readable name.
- **title** (optional, type `string`) – A human-readable variation title.
- **description** (optional, type `string`) – A human-readable variation description.
- **category** (optional, type `string`) – A category classification used in search interfaces to arrange block types by category.
- **keywords** (optional, type `string[]`) – An array of terms (which can be translated) that help users discover the variation while searching.
- **icon** (optional, type `string | Object`) – An icon helping to visualize the variation. It can have the same shape as the block type.
- **attributes** (optional, type `Object`) – Values that override block attributes.
- **innerBlocks** (optional, type `Array[]`) – Initial configuration of nested blocks.
- **example** (optional, type `Object`) – Provides structured data for the block preview. Set to `undefined` to disable the preview. See the [Block Registration API](#) for more details.
- **scope** (optional, type `WPBlockVariationScope[]`) – Defaults to `block` and `inserter`. The list of scopes where the variation is applicable. Available options include:
 - `block` – Used by blocks to filter specific block variations. `Columns` and `Query` blocks have such variations, which are passed to the [experimental BlockVariationPicker](#) component. This component handles displaying the variations and allows users to choose one of them.
 - `inserter` – Block variation is shown on the inserter.
 - `transform` – Block variation is shown in the component for variation transformations.
- **isDefault** (optional, type `boolean`) – Defaults to `false`. Indicates whether the current variation is the default one (details below).
- **isActive** (optional, type `Function | string[]`) – A function or an array of block attributes that is used to determine if the variation is active when the block is selected. The function accepts `blockAttributes` and `variationAttributes` (details below).

You can technically create a block variation without a unique `name`, but this is **not** recommended. A unique `name` allows the Editor to differentiate between your variation and others that may exist. It also allows your variation to be unregistered as needed and has implications for the `isDefault` settings (details below).

Creating a block variation

Block variations can be declared during a block's registration by providing the `variations` key with a proper array of variation objects, as shown in the example above. See the [Block Registration API](#) for more details.

To create a variation for an existing block, such as a Core block, use `wp.blocks.registerBlockVariation()`. This function accepts the name of the block and the object defining the variation.

```
wp.blocks.registerBlockVariation(
  'core/embed',
  {
    name: 'custom-embed',
    attributes: { providerNameSlug: 'custom' },
  }
);
```

Removing a block variation

Block variations can also be easily removed. To do so, use `wp.blocks.unregisterBlockVariation()`. This function accepts the name of the block and the name of the variation that should be unregistered.

```
wp.blocks.unregisterBlockVariation( 'core/embed', 'youtube' );
```

Block variations versus block styles

The main difference between block styles and block variations is that a block style just applies a CSS class to the block, so it can be styled in an alternative way. See the [Block Styles API](#) for more details.

If you want to apply initial attributes or inner blocks, this falls into block variation territory. It's also possible to override the default block style using the `className` attribute when defining a block variation.

```
variations: [
  {
    name: 'blue',
    title: __( 'Blue Quote' ),
    isDefault: true,
    attributes: {
      color: 'blue',
      className: 'is-style-blue-quote'
    },
    icon: 'format-quote',
    isActive: ( blockAttributes, variationAttributes ) =>
      blockAttributes.color === variationAttributes.color
  },
],
```

Using isDefault

By default, all variations will show up in the Inserter in addition to the original block type item. However, setting the `isDefault` flag for any variations listed will override the regular block type in the Inserter. This is a great tool for curating the Editor experience to your specific needs.

For example, if you want Media & Text block to display the image on the right by default, you could create a variation like this:

```
wp.blocks.registerBlockVariation(
  'core/media-text',
  {
    name: 'media-text-media-right',
    title: __( 'Media & Text' ),
    isDefault: true,
    attributes: {
      mediaPosition: 'right'
    }
  }
)
```

Caveats to using isDefault

While `isDefault` works great when overriding a block without existing variations, you may run into issues when other variations exist.

If another variation for the same block uses `isDefault`, your variation will not necessarily become the default. The Editor respects the first registered variation with `isDefault`, which might not be yours.

The solution is to unregister the other variation before registering your variation with `isDefault`. This caveat reinforces the recommendation always to provide variations with a unique name. Otherwise, the variation cannot be unregistered.

Using isActive

While the `isActive` property is optional, you will often want to use it to display information about the block variation after the block has been inserted. For example, this API is used in `useBlockDisplayInformation` hook to fetch and display proper information in places like the `BlockCard` or `Breadcrumbs` components.

If `isActive` is not set, the Editor cannot distinguish between the original block and your variation, so the original block information will be displayed.

The property can use either a function or an array of strings (`string[]`). The function accepts `blockAttributes` and `variationAttributes`, which can be used to determine if a variation is active. In the Embed block, the primary differentiator is the `providerNameSlug` attribute, so if you wanted to determine if the YouTube Embed variation was active, you could do something like this:

```
isActive: ( blockAttributes, variationAttributes ) =>
  blockAttributes.providerNameSlug === variationAttributes.providerNameS
```

You can also use a `string` [] to tell which attributes should be compared as a shorthand. Each attribute will be checked and the variation will be active if all of them match. Using the same example of the YouTube Embed variation, the string version would look like this:

```
isActive: [ 'providerNameSlug' ]
```

Caveats to using isActive

The `isActive` property can return false positives if multiple variations exist for a specific block and the `isActive` checks are not specific enough. To demonstrate this, consider the following example:

```
wp.blocks.registerBlockVariation(
  'core/paragraph',
  {
    name: 'paragraph-red',
    title: 'Red Paragraph',
    attributes: {
      textColor: 'vivid-red',
    },
    isActive: [ 'textColor' ],
  }
);

wp.blocks.registerBlockVariation(
  'core/paragraph',
  {
    name: 'paragraph-red-grey',
    title: 'Red/Grey Paragraph',
    attributes: {
      textColor: 'vivid-red',
      backgroundColor: 'cyan-bluish-gray'
    },
    isActive: [ 'textColor', 'backgroundColor' ]
  }
);
```

The `isActive` check on both variations tests the `textColor`, but each variations uses `vivid-red`. Since the `paragraph-red` variation is registered first, once the `paragraph-red-grey` variation is inserted into the Editor, it will have the title `Red Paragraph` instead of `Red/Grey Paragraph`. As soon as the Editor finds a match, it stops checking.

There have been [discussions](#) around how the API can be improved, but as of WordPress 6.3, this remains an issue to watch out for.

First published

April 21, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: Variations”](#)

[Previous Transforms](#) [Previous: Transforms](#)

[Next Core Blocks Reference](#) [Next: Core Blocks Reference](#)

Core Blocks Reference

In this article

Table of Contents

- [Archives](#)
- [Audio](#)
- [Avatar](#)
- [Pattern](#)
- [Button](#)
- [Buttons](#)
- [Calendar](#)
- [Categories List](#)
- [Code](#)
- [Column](#)
- [Columns](#)
- [Comment Author Avatar \(deprecated\)](#)
- [Comment Author Name](#)
- [Comment Content](#)
- [Comment Date](#)
- [Comment Edit Link](#)
- [Comment Reply Link](#)
- [Comment Template](#)
- [Comments](#)
- [Comments Pagination](#)
- [Comments Next Page](#)
- [Comments Page Numbers](#)
- [Comments Previous Page](#)
- [Comments Title](#)
- [Cover](#)
- [Details](#)
- [Embed](#)
- [File](#)
- [Footnotes](#)
- [Form](#)
- [Input Field](#)
- [Form Submission Notification](#)
- [Form Submit Button](#)
- [Classic](#)
- [Gallery](#)
- [Group](#)
- [Heading](#)
- [Home Link](#)
- [Custom HTML](#)
- [Image](#)
- [Latest Comments](#)

- [Latest Posts](#)
- [List](#)
- [List item](#)
- [Login/out](#)
- [Media & Text](#)
- [Unsupported](#)
- [More](#)
- [Navigation](#)
- [Custom Link](#)
- [Submenu](#)
- [Page Break](#)
- [Page List](#)
- [Page List Item](#)
- [Paragraph](#)
- [Pattern placeholder](#)
- [Author](#)
- [Author Biography](#)
- [Author Name](#)
- [Comment \(deprecated\)](#)
- [Comments Count](#)
- [Comments Form](#)
- [Comments Link](#)
- [Content](#)
- [Date](#)
- [Excerpt](#)
- [Featured Image](#)
- [Post Navigation Link](#)
- [Post Template](#)
- [Post Terms](#)
- [Time To Read](#)
- [Title](#)
- [Preformatted](#)
- [Pullquote](#)
- [Query Loop](#)
- [No results](#)
- [Pagination](#)
- [Next Page](#)
- [Page Numbers](#)
- [Previous Page](#)
- [Query Title](#)
- [Quote](#)
- [Read More](#)
- [RSS](#)
- [Search](#)
- [Separator](#)
- [Shortcode](#)
- [Site Logo](#)
- [Site Tagline](#)
- [Site Title](#)
- [Social Icon](#)
- [Social Icons](#)
- [Spacer](#)
- [Table](#)
- [Table of Contents](#)

- [Tag Cloud](#)
- [Template Part](#)
- [Term Description](#)
- [Text Columns \(deprecated\)](#)
- [Verse](#)
- [Video](#)

[↑ Back to top](#)

This page lists the blocks included in the block-library package.

- Items marked with a strikeout (~~strikeout~~) are explicitly disabled.
- Blocks marked with **Experimental:** true are only available when Gutenberg is active.
- Blocks marked with **Experimental:** fse are only available in the Site Editor.

[Archives](#)

Display a date archive of your posts. ([Source](#))

- **Name:** core/archives
- **Category:** widgets
- **Supports:** align, spacing (margin, padding), typography (fontSize, lineHeight), ~~html~~
- **Attributes:** displayAsDropdown, showLabel, showPostCounts, type

[Audio](#)

Embed a simple audio player. ([Source](#))

- **Name:** core/audio
- **Category:** media
- **Supports:** align, anchor, spacing (margin, padding)
- **Attributes:** autoplay, caption, id, loop, preload, src

[Avatar](#)

Add a user's avatar. ([Source](#))

- **Name:** core/avatar
- **Category:** theme
- **Supports:** align, color (background, text), spacing (margin, padding), alignWide, ~~html~~
- **Attributes:** isLink, linkTarget, size, userId

[Pattern](#)

Reuse this design across your site. ([Source](#))

- **Name:** core/block
- **Category:** reusable
- **Supports:** customClassName, html, inserter, renaming
- **Attributes:** overrides, ref

Button

Prompt visitors to take action with a button-style link. ([Source](#))

- **Name:** core/button
- **Category:** design
- **Parent:** core/buttons
- **Supports:** anchor, color (background, gradients, text), shadow, spacing (padding), typography (fontSize, lineHeight), alignWide, align, reusable
- **Attributes:** backgroundColor, gradient, linkTarget, placeholder, rel, tagName, text, textAlign, textColor, title, type, url, width

Buttons

Prompt visitors to take action with a group of button-style links. ([Source](#))

- **Name:** core/buttons
- **Category:** design
- **Supports:** align (full, wide), anchor, layout (default, allowInheriting, allowSwitching), spacing (blockGap, margin), typography (fontSize, lineHeight), html
- **Attributes:**

Calendar

A calendar of your site's posts. ([Source](#))

- **Name:** core/calendar
- **Category:** widgets
- **Supports:** align, color (background, link, text), typography (fontSize, lineHeight)
- **Attributes:** month, year

Categories List

Display a list of all categories. ([Source](#))

- **Name:** core/categories
- **Category:** widgets
- **Supports:** align, spacing (margin, padding), typography (fontSize, lineHeight), html
- **Attributes:** displayAsDropdown, showEmpty, showHierarchy, showOnlyTopLevel, showPostCounts

Code

Display code snippets that respect your spacing and tabs. ([Source](#))

- **Name:** core/code
- **Category:** text
- **Supports:** align (wide), anchor, color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** content

Column

A single column within a columns block. ([Source](#))

- **Name:** core/column
- **Category:** design
- **Parent:** core/columns
- **Supports:** anchor, color (background, button, gradients, heading, link, text), layout, spacing (blockGap, padding), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** allowedBlocks, templateLock, verticalAlignment, width

Columns

Display content in multiple columns, with blocks added to each column. ([Source](#))

- **Name:** core/columns
- **Category:** design
- **Supports:** align (full, wide), anchor, color (background, button, gradients, heading, link, text), layout (default, `allowEditing`, `allowInheriting`, `allowSwitching`), spacing (blockGap, margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** isStackedOnMobile, templateLock, verticalAlignment

Comment Author Avatar (deprecated)

This block is deprecated. Please use the Avatar block instead. ([Source](#))

- **Name:** core/comment-author-avatar
- **Experimental:** fse
- **Category:** theme
- **Supports:** color (background, `text`), spacing (margin, padding), `html`, `inserter`
- **Attributes:** height, width

Comment Author Name

Displays the name of the author of the comment. ([Source](#))

- **Name:** core/comment-author-name
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** isLink, linkTarget, textAlign

Comment Content

Displays the contents of a comment. ([Source](#))

- **Name:** core/comment-content
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (padding), typography (fontSize, lineHeight), `html`
- **Attributes:** textAlign

Comment Date

Displays the date on which the comment was posted. ([Source](#))

- **Name:** core/comment-date
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** format, isLink

Comment Edit Link

Displays a link to edit the comment in the WordPress Dashboard. This link is only visible to users with the edit comment capability. ([Source](#))

- **Name:** core/comment-edit-link
- **Category:** theme
- **Supports:** color (background, gradients, link, `text`), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** linkTarget, textAlign

Comment Reply Link

Displays a link to reply to a comment. ([Source](#))

- **Name:** core/comment-reply-link
- **Category:** theme
- **Supports:** color (background, gradients, link, `text`), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** textAlign

Comment Template

Contains the block elements used to display a comment, like the title, date, author, avatar and more. ([Source](#))

- **Name:** core/comment-template
- **Category:** design
- **Parent:** core/comments
- **Supports:** align, spacing (margin, padding), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:**

Comments

An advanced block that allows displaying post comments using different visual configurations. ([Source](#))

- **Name:** core/comments
- **Category:** theme

- **Supports:** align (full, wide), color (background, gradients, heading, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** legacy, tagName

Comments Pagination

Displays a paginated navigation to next/previous set of comments, when applicable. ([Source](#))

- **Name:** core/comments-pagination
- **Category:** theme
- **Parent:** core/comments
- **Supports:** align, color (background, gradients, link, text), layout (default, `allowInheriting`, `allowSwitching`), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** paginationArrow

Comments Next Page

Displays the next comment's page link. ([Source](#))

- **Name:** core/comments-pagination-next
- **Category:** theme
- **Parent:** core/comments-pagination
- **Supports:** color (background, gradients, `text`), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** label

Comments Page Numbers

Displays a list of page numbers for comments pagination. ([Source](#))

- **Name:** core/comments-pagination-numbers
- **Category:** theme
- **Parent:** core/comments-pagination
- **Supports:** color (background, gradients, `text`), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:**

Comments Previous Page

Displays the previous comment's page link. ([Source](#))

- **Name:** core/comments-pagination-previous
- **Category:** theme
- **Parent:** core/comments-pagination
- **Supports:** color (background, gradients, `text`), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** label

Comments Title

Displays a title with the number of comments. ([Source](#))

- **Name:** core/comments-title
- **Category:** theme
- **Supports:** align, color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight), anchor, html
- **Attributes:** level, showCommentsCount, showPostTitle, textAlign

Cover

Add an image or video with a text overlay. ([Source](#))

- **Name:** core/cover
- **Category:** media
- **Supports:** align, anchor, color (heading, text, background, enableContrastChecker), dimensions (aspectRatio), layout (allowJustification), spacing (blockGap, margin, padding), typography (fontSize, lineHeight), html
- **Attributes:** allowedBlocks, alt, backgroundType, contentPosition, customGradient, customOverlayColor, dimRatio, focalPoint, gradient, hasParallax, id, isDark, isRepeated, isUserOverlayColor, minHeight, minHeightUnit, overlayColor, tagName, templateLock, url, useFeaturedImage

Details

Hide and show additional content. ([Source](#))

- **Name:** core/details
- **Category:** text
- **Supports:** align (full, wide), color (background, gradients, link, text), layout (allowEditing), spacing (blockGap, margin, padding), typography (fontSize, lineHeight), html
- **Attributes:** showContent, summary

Embed

Add a block that displays content pulled from other sites, like Twitter or YouTube. ([Source](#))

- **Name:** core/embed
- **Category:** embed
- **Supports:** align, spacing (margin)
- **Attributes:** allowResponsive, caption, previewable, providerNameSlug, responsive, type, url

File

Add a link to a downloadable file. ([Source](#))

- **Name:** core/file
- **Category:** media

- **Supports:** align, anchor, color (background, gradients, link, text), interactivity, spacing (margin, padding)
- **Attributes:** displayPreview, downloadButtonText, fileId, fileName, href, id, previewHeight, showDownloadButton, textLinkHref, textLinkTarget

Footnotes

Display footnotes added to the page. ([Source](#))

- **Name:** core/footnotes
- **Category:** text
- **Supports:** color (background, link, text), spacing (margin, padding), typography (fontSize, lineHeight), html, inserter, multiple, reusable
- **Attributes:**

Form

A form. ([Source](#))

- **Name:** core/form
- **Experimental:** true
- **Category:** common
- **Supports:** anchor, color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), className
- **Attributes:** action, email, method, submissionMethod

Input Field

The basic building block for forms. ([Source](#))

- **Name:** core/form-input
- **Experimental:** true
- **Category:** common
- **Supports:** anchor, spacing (margin), reusable
- **Attributes:** inlineLabel, label, name, placeholder, required, type, value, visibilityPermissions

Form Submission Notification

Provide a notification message after the form has been submitted. ([Source](#))

- **Name:** core/form-submission-notification
- **Experimental:** true
- **Category:** common
- **Supports:**
- **Attributes:** type

Form Submit Button

A submission button for forms. ([Source](#))

- **Name:** core/form-submit-button
- **Experimental:** true
- **Category:** common
- **Supports:**
- **Attributes:**

Classic

Use the classic WordPress editor. ([Source](#))

- **Name:** core/freeform
- **Category:** text
- **Supports:** className, customClassName, reusable
- **Attributes:** content

Gallery

Display multiple images in a rich gallery. ([Source](#))

- **Name:** core/gallery
- **Category:** media
- **Supports:** align, anchor, color (background, gradients, text), layout (default, allowEditing, allowInheriting, allowSwitching), spacing (blockGap, margin, padding), units (em, px, rem, vh, vw), html
- **Attributes:** allowResize, caption, columns, fixedHeight, ids, imageCrop, images, linkTarget, linkTo, randomOrder, shortCodeTransforms, sizeSlug

Group

Gather blocks in a layout container. ([Source](#))

- **Name:** core/group
- **Category:** design
- **Supports:** align (full, wide), anchor, ariaLabel, background (backgroundImage, backgroundSize), color (background, button, gradients, heading, link, text), dimensions (minHeight), layout (allowSizingOnChildren), position (sticky), spacing (blockGap, margin, padding), typography (fontSize, lineHeight), html
- **Attributes:** allowedBlocks, tagName, templateLock

Heading

Introduce new sections and organize content to help visitors (and search engines) understand the structure of your content. ([Source](#))

- **Name:** core/heading
- **Category:** text

- **Supports:** __unstablePasteTextInline, align (full, wide), anchor, className, color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** content, level, placeholder, textAlign

Home Link

Create a link that always points to the homepage of the site. Usually not necessary if there is already a site title link present in the header. ([Source](#))

- **Name:** core/home-link
- **Category:** design
- **Parent:** core/navigation
- **Supports:** typography (fontSize, lineHeight), html, reusable
- **Attributes:** label

Custom HTML

Add custom HTML code and preview it as you edit. ([Source](#))

- **Name:** core/html
- **Category:** widgets
- **Supports:** eClassName, ecustomClassName, html
- **Attributes:** content

Image

Insert an image to make a visual statement. ([Source](#))

- **Name:** core/image
- **Category:** media
- **Supports:** align (center, full, left, right, wide), anchor, color (background, text), filter (duotone), interactivity
- **Attributes:** alt, aspectRatio, caption, height, href, id, lightbox, linkClass, linkDestination, linkTarget, rel, scale, sizeSlug, title, url, width

Latest Comments

Display a list of your most recent comments. ([Source](#))

- **Name:** core/latest-comments
- **Category:** widgets
- **Supports:** align, spacing (margin, padding), typography (fontSize, lineHeight), html
- **Attributes:** commentsToShow, displayAvatar, displayDate, displayExcerpt

Latest Posts

Display a list of your most recent posts. ([Source](#))

- **Name:** core/latest-posts
- **Category:** widgets

- **Supports:** align, color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** addLinkToFeaturedImage, categories, columns, displayAuthor, displayFeaturedImage, displayPostContent, displayPostContentRadio, displayPostDate, excerptLength, featuredImageAlign, featuredImageSizeHeight, featuredImageSizeSlug, featuredImageSizeWidth, order, orderBy, postLayout, postsToShow, selectedAuthor

List

Create a bulleted or numbered list. ([Source](#))

- **Name:** core/list
- **Category:** text
- **Supports:** __unstablePasteTextInline, anchor, color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `eClassName`
- **Attributes:** ordered, placeholder, reversed, start, type, values

List item

Create a list item. ([Source](#))

- **Name:** core/list-item
- **Category:** text
- **Parent:** core/list
- **Supports:** spacing (margin, padding), typography (fontSize, lineHeight), `eClassName`
- **Attributes:** content, placeholder

Login/out

Show login & logout links. ([Source](#))

- **Name:** core/loginout
- **Category:** theme
- **Supports:** className, spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** displayLoginAsForm, redirectToCurrent

Media & Text

Set media and words side-by-side for a richer layout. ([Source](#))

- **Name:** core/media-text
- **Category:** media
- **Supports:** align (full, wide), anchor, color (background, gradients, heading, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** align, allowedBlocks, focalPoint, href, imageFill, isStackedOnMobile, linkClass, linkDestination, linkTarget, mediaAlt, mediaId, mediaLink, mediaPosition, mediaSizeSlug, mediaType, mediaUrl, mediaWidth, rel, verticalAlignment

Unsupported

Your site doesn't include support for this block. ([Source](#))

- **Name:** core/missing
- **Category:** text
- **Supports:** className, customClassName, html, inserter, reusable
- **Attributes:** originalContent, originalName, originalUndelimitedContent

More

Content before this block will be shown in the excerpt on your archives page. ([Source](#))

- **Name:** core/more
- **Category:** design
- **Supports:** className, customClassName, html, multiple
- **Attributes:** customText, noTeaser

Navigation

A collection of blocks that allow visitors to get around your site. ([Source](#))

- **Name:** core/navigation
- **Category:** theme
- **Supports:** align (full, wide), ariaLabel, inserter, interactivity, layout (allowSizingOnChildren, default, allowInheriting, allowSwitching, allowVerticalAlignment), spacing (blockGap, units), typography (fontSize, lineHeight), html, renaming
- **Attributes:** __unstableLocation, backgroundColor, customBackgroundColor, customOverlayBackgroundColor, customOverlayTextColor, customTextColor, hasIcon, icon, maxNestingLevel, openSubmenusOnClick, overlayBackgroundColor, overlayMenu, overlayTextColor, ref, rgbBackgroundColor, rgbTextColor, show_submenuIcon, templateLock, textColor

Custom Link

Add a page, link, or another item to your navigation. ([Source](#))

- **Name:** core/navigation-link
- **Category:** design
- **Parent:** core/navigation
- **Supports:** typography (fontSize, lineHeight), html, renaming, reusable
- **Attributes:** description, id, isTopLevelLink, kind, label, opensInNewTab, rel, title, type, url

Submenu

Add a submenu to your navigation. ([Source](#))

- **Name:** core/navigation-submenu
- **Category:** design

- **Parent:** core/navigation
- **Supports:** html, reusable
- **Attributes:** description, id, isTopLevelItem, kind, label, opensInNewTab, rel, title, type, url

Page Break

Separate your content into a multi-page experience. ([Source](#))

- **Name:** core/nextpage
- **Category:** design
- **Parent:** core/post-content
- **Supports:** className, customClassName, html
- **Attributes:**

Page List

Display a list of all pages. ([Source](#))

- **Name:** core/page-list
- **Category:** widgets
- **Supports:** typography (fontSize, lineHeight), html, reusable
- **Attributes:** isNested, parentPageID

Page List Item

Displays a page inside a list of all pages. ([Source](#))

- **Name:** core/page-list-item
- **Category:** widgets
- **Parent:** core/page-list
- **Supports:** html, inserter, lock, reusable
- **Attributes:** hasChildren, id, label, link, title

Paragraph

Start with the basic building block of all narrative. ([Source](#))

- **Name:** core/paragraph
- **Category:** text
- **Supports:** __unstablePasteTextInline, anchor, color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), className
- **Attributes:** align, content, direction, dropCap, placeholder

Pattern placeholder

Show a block pattern. ([Source](#))

- **Name:** core/pattern
- **Category:** theme
- **Supports:** html, inserter, renaming
- **Attributes:** slug

Author

Display post author details such as name, avatar, and bio. ([Source](#))

- **Name:** core/post-author
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** avatarSize, byline, isLink, linkTarget, showAvatar, showBio, textAlign

Author Biography

The author biography. ([Source](#))

- **Name:** core/post-author-biography
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** textAlign

Author Name

The author name. ([Source](#))

- **Name:** core/post-author-name
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** isLink, linkTarget, textAlign

Comment (deprecated)

This block is deprecated. Please use the Comments block instead. ([Source](#))

- **Name:** core/post-comment
- **Experimental:** fse
- **Category:** theme
- **Supports:** `html`, `inserter`
- **Attributes:** commentId

Comments Count

Display a post's comments count. ([Source](#))

- **Name:** core/post-comments-count
- **Experimental:** fse
- **Category:** theme
- **Supports:** color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** textAlign

Comments Form

Display a post's comments form. ([Source](#))

- **Name:** core/post-comments-form
- **Category:** theme
- **Supports:** color (background, gradients, heading, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** textAlign

Comments Link

Displays the link to the current post comments. ([Source](#))

- **Name:** core/post-comments-link
- **Experimental:** fse
- **Category:** theme
- **Supports:** color (background, link, `text`), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** textAlign

Content

Displays the contents of a post or page. ([Source](#))

- **Name:** core/post-content
- **Category:** theme
- **Supports:** align (full, wide), color (background, gradients, link, text), dimensions (minHeight), layout, spacing (blockGap), typography (fontSize, lineHeight), `html`
- **Attributes:**

Date

Display the publish date for an entry such as a post or page. ([Source](#))

- **Name:** core/post-date
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** displayType, format, isLink, textAlign

Excerpt

Display the excerpt. ([Source](#))

- **Name:** core/post-excerpt
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** excerptLength, moreText, showMoreOnNewLine, textAlign

Featured Image

Display a post's featured image. ([Source](#))

- **Name:** core/post-featured-image
- **Category:** theme
- **Supports:** align (center, full, left, right, wide), color (background, text), spacing (margin, padding), `html`
- **Attributes:** aspectRatio, customGradient, customOverlayColor, dimRatio, gradient, height, isLink, linkTarget, overlayColor, rel, scale, sizeSlug, useFirstImageFromPost, width

Post Navigation Link

Displays the next or previous post link that is adjacent to the current post. ([Source](#))

- **Name:** core/post-navigation-link
- **Category:** theme
- **Supports:** color (background, link, text), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** arrow, label,.linkLabel, showTitle, taxonomy, textAlign, type

Post Template

Contains the block elements used to render a post, like the title, date, featured image, content or excerpt, and more. ([Source](#))

- **Name:** core/post-template
- **Category:** theme
- **Parent:** core/query
- **Supports:** align (full, wide), color (background, gradients, link, text), layout, spacing (blockGap), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:**

Post Terms

Post terms. ([Source](#))

- **Name:** core/post-terms
- **Category:** theme
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** prefix, separator, suffix, term, textAlign

Time To Read

Show minutes required to finish reading the post. ([Source](#))

- **Name:** core/post-time-to-read
- **Experimental:** true
- **Category:** theme
- **Supports:** color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`

- **Attributes:** textAlign

Title

Displays the title of a post, page, or any other content-type. ([Source](#))

- **Name:** core/post-title
- **Category:** theme
- **Supports:** align (full, wide), color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** isLink, level, linkTarget, rel, textAlign

Preformatted

Add text that respects your spacing and tabs, and also allows styling. ([Source](#))

- **Name:** core/preformatted
- **Category:** text
- **Supports:** anchor, color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** content

Pullquote

Give special visual emphasis to a quote from your text. ([Source](#))

- **Name:** core/pullquote
- **Category:** text
- **Supports:** align (full, left, right, wide), anchor, color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** citation, textAlign, value

Query Loop

An advanced block that allows displaying post types based on different query parameters and visual configurations. ([Source](#))

- **Name:** core/query
- **Category:** theme
- **Supports:** align (full, wide), interactivity, layout, `html`
- **Attributes:** enhancedPagination, namespace, query, queryId, tagName

No results

Contains the block elements used to render content when no query results are found. ([Source](#))

- **Name:** core/query-no-results
- **Category:** theme
- **Parent:** core/query
- **Supports:** align, color (background, gradients, link, text), typography (fontSize, lineHeight), `html`, `reusable`

- **Attributes:**

Pagination

Displays a paginated navigation to next/previous set of posts, when applicable. ([Source](#))

- **Name:** core/query-pagination
- **Category:** theme
- **Parent:** core/query
- **Supports:** align, color (background, gradients, link, text), layout (default, allowInheriting, allowSwitching), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** paginationArrow, showLabel

Next Page

Displays the next posts page link. ([Source](#))

- **Name:** core/query-pagination-next
- **Category:** theme
- **Parent:** core/query-pagination
- **Supports:** color (background, gradients, `text`), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** label

Page Numbers

Displays a list of page numbers for pagination. ([Source](#))

- **Name:** core/query-pagination-numbers
- **Category:** theme
- **Parent:** core/query-pagination
- **Supports:** color (background, gradients, `text`), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** midSize

Previous Page

Displays the previous posts page link. ([Source](#))

- **Name:** core/query-pagination-previous
- **Category:** theme
- **Parent:** core/query-pagination
- **Supports:** color (background, gradients, `text`), typography (fontSize, lineHeight), `html`, `reusable`
- **Attributes:** label

Query Title

Display the query title. ([Source](#))

- **Name:** core/query-title

- **Name:** core/quote
- **Category:** theme
- **Supports:** align (full, wide), color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** level, showPrefix, showSearchTerm, textAlign, type

Quote

Give quoted text visual emphasis. “In quoting others, we cite ourselves.” — Julio Cortázar ([Source](#))

- **Name:** core/read-more
- **Category:** text
- **Supports:** anchor, color (background, gradients, heading, link, text), layout (`allowEditing`), spacing (blockGap), typography (fontSize, lineHeight), `html`
- **Attributes:** align, citation, value

Read More

Displays the link of a post, page, or any other content-type. ([Source](#))

- **Name:** core/rss
- **Category:** theme
- **Supports:** color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** content, linkTarget

RSS

Display entries from any RSS or Atom feed. ([Source](#))

- **Name:** core/rss
- **Category:** widgets
- **Supports:** align, `html`
- **Attributes:** blockLayout, columns, displayAuthor, displayDate, displayExcerpt, excerptLength, feedURL, itemsToShow

Search

Help visitors find your content. ([Source](#))

- **Name:** core/search
- **Category:** widgets
- **Supports:** align (center, left, right), color (background, gradients, text), interactivity, typography (fontSize, lineHeight), `html`
- **Attributes:** buttonPosition, buttonText, buttonUseIcon, isSearchFieldHidden, label, placeholder, query, showLabel, width, widthUnit

Separator

Create a break between ideas or sections with a horizontal separator. ([Source](#))

- **Name:** core/separator
- **Category:** design
- **Supports:** align (center, full, wide), anchor, color (background, gradients, enableContrastChecker, text), spacing (margin)
- **Attributes:** opacity

Shortcode

Insert additional custom elements with a WordPress shortcode. ([Source](#))

- **Name:** core/shortcode
- **Category:** widgets
- **Supports:** className, customClassName, html
- **Attributes:** text

Site Logo

Display an image to represent this site. Update this block and the changes apply everywhere. ([Source](#))

- **Name:** core/site-logo
- **Category:** theme
- **Supports:** align, color (background, text), spacing (margin, padding), alignWide, html
- **Attributes:** isLink, linkTarget, shouldSyncIcon, width

Site Tagline

Describe in a few words what the site is about. The tagline can be used in search results or when sharing on social networks even if it's not displayed in the theme design. ([Source](#))

- **Name:** core/site-tagline
- **Category:** theme
- **Supports:** align (full, wide), color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight), html
- **Attributes:** textAlign

Site Title

Displays the name of this site. Update the block, and the changes apply everywhere it's used. This will also appear in the browser title bar and in search results. ([Source](#))

- **Name:** core/site-title
- **Category:** theme
- **Supports:** align (full, wide), color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), html
- **Attributes:** isLink, level, linkTarget, textAlign

Social Icon

Display an icon linking to a social media profile or site. ([Source](#))

- **Name:** core/social-link
- **Category:** widgets
- **Parent:** core/social-links
- **Supports:** html, reusable
- **Attributes:** label, rel, service, url

Social Icons

Display icons linking to your social media profiles or sites. ([Source](#))

- **Name:** core/social-links
- **Category:** widgets
- **Supports:** align (center, left, right), anchor, color (background, gradients, enableContrastChecker, text), layout (default, allowInheriting, allowSwitching, allowVerticalAlignment), spacing (blockGap, margin, padding, units)
- **Attributes:** customIconBackgroundColor, customIconColor, iconBackgroundColor, iconBackgroundColorValue, iconColor, iconColorValue, openInNewTab, showLabels, size

Spacer

Add white space between blocks and customize its height. ([Source](#))

- **Name:** core/spacer
- **Category:** design
- **Supports:** anchor, spacing (margin)
- **Attributes:** height, width

Table

Create structured content in rows and columns to display information. ([Source](#))

- **Name:** core/table
- **Category:** text
- **Supports:** align, anchor, color (background, gradients, text), spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** body, caption, foot, hasFixedLayout, head

Table of Contents

Summarize your post with a list of headings. Add HTML anchors to Heading blocks to link them here. ([Source](#))

- **Name:** core/table-of-contents
- **Experimental:** true
- **Category:** layout
- **Supports:** color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight), html

- **Attributes:** headings, onlyIncludeCurrentPage

Tag Cloud

A cloud of your most used tags. ([Source](#))

- **Name:** core/tag-cloud
- **Category:** widgets
- **Supports:** align, spacing (margin, padding), typography (lineHeight), `html`
- **Attributes:** largestFontSize, numberOfTags, showTagCounts, smallestFontSize, taxonomy

Template Part

Edit the different global regions of your site, like the header, footer, sidebar, or create your own. ([Source](#))

- **Name:** core/template-part
- **Category:** theme
- **Supports:** align, `html`, renaming, reusable
- **Attributes:** area, slug, tagName, theme

Term Description

Display the description of categories, tags and custom taxonomies when viewing an archive. ([Source](#))

- **Name:** core/term-description
- **Category:** theme
- **Supports:** align (full, wide), color (background, link, text), spacing (margin, padding), typography (fontSize, lineHeight), `html`
- **Attributes:** textAlign

Text Columns (deprecated)

This block is deprecated. Please use the Columns block instead. ([Source](#))

- **Name:** core/text-columns
- **Category:** design
- **Supports:** inserter
- **Attributes:** columns, content, width

Verse

Insert poetry. Use special spacing formats. Or quote song lyrics. ([Source](#))

- **Name:** core/verse
- **Category:** text
- **Supports:** anchor, color (background, gradients, link, text), spacing (margin, padding), typography (fontSize, lineHeight)
- **Attributes:** content, textAlign

Video

Embed a video from your media library or upload a new one. ([Source](#))

- **Name:** core/video
- **Category:** media
- **Supports:** align, anchor, spacing (margin, padding)
- **Attributes:** autoplay, caption, controls, id, loop, muted, playsInline, poster, preload, src, tracks

First published

December 20, 2021

Last updated

January 30, 2024

Edit article

[Improve it on GitHub: Core Blocks Reference”](#)

[Previous Variations](#) [Previous: Variations](#)

[Next Hooks Reference](#) [Next: Hooks Reference](#)

Hooks Reference

[↑ Back to top](#)

[Hooks](#) are a way for one piece of code to interact/modify another piece of code. They provide one way for plugins and themes to interact with the editor, but they’re also used extensively by WordPress Core itself.

There are two types of hooks: [Actions](#) and [Filters](#). In addition to PHP actions and filters, WordPress also provides a mechanism for registering and executing hooks in JavaScript. This functionality is also available on npm as the [@wordpress/hooks](#) package, for general purpose use.

You can also learn more about both APIs: [PHP](#) and [JavaScript](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Hooks Reference”](#)

[Previous Core Blocks Reference](#) [Previous: Core Blocks Reference](#)

Block Filters

In this article

Table of Contents

- [Registration](#)
 - [block_type_metadata](#)
 - [block_type_metadata_settings](#)
 - [blocks.registerBlockType](#)
- [Block Editor](#)
 - [blocks.getSaveElement](#)
 - [blocks.getSaveContent.extraProps](#)
 - [blocks.getBlockDefaultClassName](#)
 - [blocks.switchToBlockType.transformedBlock](#)
 - [blocks.getBlockAttributes](#)
 - [editor.BlockEdit](#)
 - [editor.BlockListBlock](#)
- [Removing Blocks](#)
 - [Using a deny list](#)
 - [Using an allow list](#)
- [Hiding blocks from the inserter](#)
 - [allowed_block_types_all](#)
- [Managing block categories](#)
 - [block_categories_all](#)
 - [wp.blocks.updateCategory](#)

[↑ Back to top](#)

To modify the behavior of existing blocks, WordPress exposes several APIs.

[Registration](#)

The following filters are available to extend the settings for blocks during their registration.

[block_type_metadata](#)

Filters the raw metadata loaded from the `block.json` file when registering a block type on the server with PHP. It allows applying modifications before the metadata gets processed.

The filter takes one param:

- `$metadata (array)` – metadata loaded from `block.json` for registering a block type.

Example:

```
<?php
```

```

function wpdocs_filter_metadata_registration( $metadata ) {
    $metadata['apiVersion'] = 1;
    return $metadata;
}
add_filter( 'block_type_metadata', 'wpdocs_filter_metadata_registration' )
register_block_type( __DIR__ );

```

block type metadata settings

Filters the settings determined from the processed block type metadata. It makes it possible to apply custom modifications using the block metadata that isn't handled by default.

The filter takes two params:

- `$settings (array)` – Array of determined settings for registering a block type.
- `$metadata (array)` – Metadata loaded from the `block.json` file.

Example:

```

function wpdocs_filter_metadata_registration( $settings, $metadata ) {
    $settings['api_version'] = $metadata['apiVersion'] + 1;
    return $settings;
}
add_filter( 'block_type_metadata_settings', 'wpdocs_filter_metadata_registration' )
register_block_type( __DIR__ );

```

blocks.registerBlockType

Used to filter the block settings when registering the block on the client with JavaScript. It receives the block settings, the name of the registered block, and either null or the deprecated block settings (when applied to a registered deprecation) as arguments. This filter is also applied to each of a block's deprecated settings.

Example:

Ensure that List blocks are saved with the canonical generated class name (`wp-block-list`):

```

function addListBlockClassName( settings, name ) {
    if ( name !== 'core/list' ) {
        return settings;
    }

    return {
        ...settings,
        supports: {
            ...settings.supports,
            className: true,
        },
    };
}

wp.hooks.addFilter(

```

```
'blocks.registerBlockType',
'my-plugin/class-names/list-block',
addListBlockClassName
);
```

Block Editor

The following filters are available to change the behavior of blocks while editing in the block editor.

blocks.getSaveElement

A filter that applies to the result of a block's `save` function. This filter is used to replace or extend the element, for example using `React.createElement` to modify the element's props or replace its children, or returning an entirely new element.

The filter's callback receives an element, a block type definition object and the block attributes as arguments. It should return an element.

Example:

Wraps a cover block into an outer container.

```
function wrapCoverBlockInContainer( element, blockType, attributes ) {
    // skip if element is undefined
    if ( ! element ) {
        return;
    }

    // only apply to cover blocks
    if ( blockType.name !== 'core/cover' ) {
        return element;
    }

    // return the element wrapped in a div
    return <div className="cover-block-wrapper">{ element }</div>;
}

wp.hooks.addFilter(
    'blocks.getSaveElement',
    'my-plugin/wrap-cover-block-in-container',
    wrapCoverBlockInContainer
);
```

blocks.getSaveContent.extraProps

A filter that applies to all blocks returning a WP Element in the `save` function. This filter is used to add extra props to the root element of the `save` function. For example: to add a `className`, an `id`, or any valid prop for this element.

The filter receives the current `save` element's props, a block type and the block attributes as arguments. It should return a `props` object.

Example:

Adding a background by default to all blocks.

```
function addBackgroundColorStyle( props ) {
    return {
        ...props,
        style: { backgroundColor: 'red' },
    };
}

wp.hooks.addFilter(
    'blocks.getSaveContent.extraProps',
    'my-plugin/add-background-color-style',
    addBackgroundColorStyle
);
```

Note: A [block validation](#) error will occur if this filter modifies existing content the next time the post is edited. The editor verifies that the content stored in the post matches the content output by the `save()` function.

To avoid this validation error, use `render_block` server-side to modify existing post content instead of this filter. See [render_block documentation](#).

[**blocks.getBlockDefaultClassName**](#)

Generated HTML classes for blocks follow the `wp-block-{name}` nomenclature. This filter allows to provide an alternative class name.

Example:

```
// Our filter function
function setBlockCustomClassName( className, blockName ) {
    return blockName === 'core/code' ? 'my-plugin-code' : className;
}

// Adding the filter
wp.hooks.addFilter(
    'blocks.getBlockDefaultClassName',
    'my-plugin/set-block-custom-class-name',
    setBlockCustomClassName
);
```

[**blocks.switchToBlockType.transformedBlock**](#)

Used to filter an individual transform result from block transformation. All of the original blocks are passed since transformations are many-to-many, not one-to-one.

[**blocks.getBlockAttributes**](#)

Called immediately after the default parsing of a block's attributes and before validation to allow a plugin to manipulate attribute values in time for validation and/or the initial values rendering of the block in the editor.

[editor.BlockEdit](#)

Used to modify the block's `edit` component. It receives the original block `BlockEdit` component and returns a new wrapped component.

Example:

```
const { createHigherOrderComponent } = wp.compose;
const { InspectorControls } = wp.blockEditor;
const { PanelBody } = wp.components;

const withMyPluginControls = createHigherOrderComponent( ( BlockEdit ) =>
  return ( props ) => {
    return (
      <>
        <BlockEdit key="edit" { ...props } />
        <InspectorControls>
          <PanelBody>My custom control</PanelBody>
        </InspectorControls>
      </>
    );
  };
}, 'withMyPluginControls' );

wp.hooks.addFilter(
  'editor.BlockEdit',
  'my-plugin/with-inspector-controls',
  withMyPluginControls
);
```

Note that as this hook is run for *all blocks*, consuming it has potential for performance regressions particularly around block selection metrics.

To mitigate this, consider whether any work you perform can be altered to run only under certain conditions.

For example, if you are adding components that only need to render when the block is *selected*, then you can use the block's “selected” state (`props.isSelected`) to conditionalize your rendering.

```
const withMyPluginControls = createHigherOrderComponent( ( BlockEdit ) =>
  return ( props ) => {
    return (
      <>
        <BlockEdit { ...props } />
        { props.isSelected && [
          <InspectorControls>
            <PanelBody>My custom control</PanelBody>
          </InspectorControls>
        ]}
      </>
    );
  };
}, 'withMyPluginControls' );
```

[editor.BlockListBlock](#)

Used to modify the block's wrapper component containing the block's `edit` component and all toolbars. It receives the original `BlockListBlock` component and returns a new wrapped component.

Example:

```
const { createHigherOrderComponent } = wp.compose;

const withClientIdClassName = createHigherOrderComponent(
  ( BlockListBlock ) => {
    return ( props ) => {
      return (
        <BlockListBlock
          { ...props }
          className={ 'block-' + props.clientId }
        />
      );
    };
  },
  'withClientIdClassName'
);

wp.hooks.addFilter(
  'editor.BlockListBlock',
  'my-plugin/with-client-id-class-name',
  withClientIdClassName
);
```

Adding new properties to the block's wrapper component can be achieved by adding them to the `wrapperProps` property of the returned component.

Example:

```
const { createHigherOrderComponent } = wp.compose;
const withMyWrapperProp = createHigherOrderComponent( ( BlockListBlock ) =
  return ( props ) => {
    const wrapperProps = {
      ...props.wrapperProps,
      'data-my-property': 'the-value',
    };
    return <BlockListBlock { ...props } wrapperProps={ wrapperProps } >;
  },
  'withMyWrapperProp' );
wp.hooks.addFilter(
  'editor.BlockListBlock',
  'my-plugin/with-my-wrapper-prop',
  withMyWrapperProp
);
```

Removing Blocks

Using a deny list

Adding blocks is easy enough, removing them is as easy. Plugin or theme authors have the possibility to “unregister” blocks.

```
// my-plugin.js
import { unregisterBlockType } from '@wordpress/blocks';
import domReady from '@wordpress/dom-ready';

domReady( function () {
    unregisterBlockType( 'core/verse' );
} );
```

and load this script in the Editor

```
<?php
// my-plugin.php

function my_plugin_deny_list_blocks() {
    wp_enqueue_script(
        'my-plugin-deny-list-blocks',
        plugins_url( 'my-plugin.js', __FILE__ ),
        array( 'wp-blocks', 'wp-dom-ready', 'wp-edit-post' )
    );
}
add_action( 'enqueue_block_editor_assets', 'my_plugin_deny_list_blocks' );
```

Important: When unregistering a block, there can be a [race condition](#) on which code runs first: registering the block, or unregistering the block. You want your unregister code to run last. The way to do that is specify the component that is registering the block as a dependency, in this case `wp-edit-post`. Additionally, using `wp.domReady()` ensures the unregister code runs once the dom is loaded.

Using an allow list

If you want to disable all blocks except an allow list, you can adapt the script above like so:

```
// my-plugin.js

var allowedBlocks = [
    'core/paragraph',
    'core/image',
    'core/html',
    'core/freeform',
];
wp.blocks.getBlockTypes().forEach( function ( blockType ) {
    if ( allowedBlocks.indexOf( blockType.name ) === -1 ) {
        wp.blocks.unregisterBlockType( blockType.name );
    }
} );
```

Hiding blocks from the inserter

allowed_block_types_all

Note: Before WordPress 5.8 known as `allowed_block_types`. In the case when you want to support older versions of WordPress you might need a way to detect which filter should be used – the deprecated one vs the new one. The recommended way to proceed is to check if the `WP_Block_Editor_Context` class exists.

On the server, you can filter the list of blocks shown in the inserter using the `allowed_block_types_all` filter. You can return either true (all block types supported), false (no block types supported), or an array of block type names to allow. You can also use the second provided param `$editor_context` to filter block types based on its content.

```
<?php
// my-plugin.php

function wpdocs_filter_allowed_block_types_when_post_provided( $allowed_block_types ) {
    if ( ! empty( $editor_context->post ) ) {
        return array( 'core/paragraph', 'core/heading' );
    }
    return $allowed_block_types;
}

add_filter( 'allowed_block_types_all', 'wpdocs_filter_allowed_block_types_all' );
```

Managing block categories

block_categories_all

Note: Before WordPress 5.8 known as `block_categories`. In the case when you want to support older versions of WordPress you might need a way to detect which filter should be used – the deprecated one vs the new one. The recommended way to proceed is to check if the `WP_Block_Editor_Context` class exists.

It is possible to filter the list of default block categories using the `block_categories_all` filter. You can do it on the server by implementing a function which returns a list of categories. It is going to be used during blocks registration and to group blocks in the inserter. You can also use the second provided param `$editor_context` to filter the based on its content.

```
<?php
// my-plugin.php

function wpdocs_filter_block_categories_when_post_provided( $block_categories ) {
    if ( ! empty( $editor_context->post ) ) {
        array_push(
            $block_categories,
            array(
                'slug'  => 'custom-category',
                'title' => __( 'Custom Category', 'custom-plugin' ),
                'icon'  => null,
            )
        );
    }
}
```

```
        );
    }
    return $block_categories;
}

add_filter( 'block_categories_all', 'wpdocs_filter_block_categories_when_p
```

[wp.blocks.updateCategory](#)

You can also display an icon with your block category by setting an `icon` attribute. The value can be the slug of a [WordPress Dashicon](#).

You can also set a custom icon in SVG format. To do so, the icon should be rendered and set on the frontend, so it can make use of WordPress SVG, allowing mobile compatibility and making the icon more accessible.

To set an SVG icon for the category shown in the previous example, add the following example JavaScript code to the editor calling `wp.blocks.updateCategory` e.g:

```
( function () {
    var el = React.createElement;
    var SVG = wp.primitives.SVG;
    var circle = el( 'circle', {
        cx: 10,
        cy: 10,
        r: 10,
        fill: 'red',
        stroke: 'blue',
        strokeWidth: '10',
    } );
    var svgIcon = el(
        SVG,
        { width: 20, height: 20, viewBox: '0 0 20 20' },
        circle
    );
    wp.blocks.updateCategory( 'my-category', { icon: svgIcon } );
} )();
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Block Filters](#)

[Previous Hooks Reference](#) [Previous: Hooks Reference](#)

[Next Editor Hooks](#) [Next: Editor Hooks](#)

Editor Hooks

In this article

Table of Contents

- [Editor features](#)
 - [editor.PostFeaturedImage.imageSize](#)
 - [editor.PostPreview.interstitialMarkup](#)
 - [media.crossOrigin](#)
- [Editor settings](#)
 - [block_editor_settings_all](#)
 - [Available default editor settings](#)
- [Logging errors](#)
 - [editor.ErrorBoundary.errorLogged](#)
- [Block Directory](#)
- [Block Patterns](#)
 - [should_load_remote_block_patterns](#)

[↑ Back to top](#)

To modify the behavior of the editor experience, WordPress exposes several APIs.

[Editor features](#)

The following filters are available to extend the editor features.

[editor.PostFeaturedImage.imageSize](#)

Used to modify the image size displayed in the Post Featured Image component. It defaults to 'post-thumbnail', and will fail back to the full image size when the specified image size doesn't exist in the media object. It's modeled after the `admin_post_thumbnail_size` filter in the classic editor.

Example:

```
var withImageSize = function ( size, mediaId, postId ) {
    return 'large';
};

wp.hooks.addFilter(
    'editor.PostFeaturedImage.imageSize',
    'my-plugin/with-image-size',
    withImageSize
);
```

[editor.PostPreview.interstitialMarkup](#)

Filters the interstitial message shown when generating previews.

Example:

```
var customPreviewMessage = function () {
    return '<b>Post preview is being generated!</b>';
};

wp.hooks.addFilter(
    'editor.PostPreview.interstitialMarkup',
    'my-plugin/custom-preview-message',
    customPreviewMessage
);
```

[media.crossOrigin](#)

Used to set or modify the `crossOrigin` attribute for foreign-origin media elements (i.e `<audio>`, ``, `<link>`, `<script>`, `<video>`). See this [article](#) for more information the `crossOrigin` attribute, its values and how it applies to each element.

One example of it in action is in the Image block's transform feature to allow cross-origin images to be used in a `<canvas>`.

Example:

```
addFilter(
    'media.crossOrigin',
    'my-plugin/with-cors-media',
    // The callback accepts a second `mediaSrc` argument which references
    // the url to actual foreign media, useful if you want to decide
    // the value of crossOrigin based upon it.
    ( crossOrigin, mediaSrc ) => {
        if ( mediaSrc.startsWith( 'https://example.com' ) ) {
            return 'use-credentials';
        }
        return crossOrigin;
    }
);
```

[Editor settings](#)

[block_editor_settings_all](#)

Note: Before WordPress 5.8 known as `block_editor_settings`. In the case when you want to support older versions of WordPress you might need a way to detect which filter should be used – the deprecated one vs the new one. The recommended way to proceed is to check if the `WP_Block_Editor_Context` class exists.

This is a PHP filter which is applied before sending settings to the WordPress block editor.

You may find details about this filter [on its WordPress Code Reference page](#).

The filter will send any setting to the initialized Editor, which means any editor setting that is used to configure the editor at initialization can be filtered by a PHP WordPress plugin before being sent.

Example:

```
<?php
// my-plugin.php

function wpdocs_filter_block_editor_settings_when_post_provided( $editor_settings ) {
    if ( ! empty( $editor_context->post ) ) {
        $editor_settings['maxUploadFileSize'] = 12345;
    }
    return $editor_settings;
}

add_filter( 'block_editor_settings_all', 'wpdocs_filter_block_editor_settings_all' );
add_filter( 'block_editor_rest_api_preload_paths', 'wpdocs_filter_block_editor_rest_api_preload_paths' );
```

Filters the array of REST API paths that will be used to preloaded common data to use with the block editor.

Example:

```
<?php
// my-plugin.php

function wpdocs_filter_block_editor_rest_api_preload_paths_when_post_provided( $preload_paths ) {
    if ( ! empty( $editor_context->post ) ) {
        array_push( $preload_paths, array( '/wp/v2/blocks', 'OPTIONS' ) );
    }
    return $preload_paths;
}

add_filter( 'block_editor_rest_api_preload_paths', 'wpdocs_filter_block_editor_rest_api_preload_paths' );
```

[Available default editor settings](#)

`richEditingEnabled`

If it is `true` the user can edit the content using the visual editor.

It is set by default to the return value of the [`user_can_richedit`](#) function. It checks if the user can access the visual editor and whether it's supported by the user's browser.

`codeEditingEnabled`

Default `true`. Indicates whether the user can access the code editor **in addition** to the visual editor.

If set to false the user will not be able to switch between visual and code editor. The option in the settings menu will not be available and the keyboard shortcut for switching editor types will not fire.

Logging errors

Note: Since WordPress 6.1.

A JavaScript error in a part of the UI shouldn't break the whole app. To solve this problem for users, React library uses a concept of an "[error boundary](#)". Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, and display a fallback UI instead of the component tree that crashed.

editor.ErrorBoundary.errorLogged

Allows you to hook into the [Error Boundaries](#) and gives you access to the error object.

You can use this action if you want to get hold of the error object that's handled by the boundaries, i.e to send them to an external error tracking tool.

Example:

```
addAction(
    'editor.ErrorBoundary.errorLogged',
    'mu-plugin/error-capture-setup',
    ( error ) => {
        // error is the exception's error object
        ErrorCaptureTool.captureError( error );
    }
);
```

Block Directory

The Block Directory enables installing new block plugins from [WordPress.org](#). It can be disabled by removing the actions that enqueue it. In WordPress core, the function is `wp_enqueue_editor_block_directory_assets`. To remove the feature, use [remove_action](#), like this:

```
<?php
// my-plugin.php

remove_action( 'enqueue_block_editor_assets', 'wp_enqueue_editor_block_dir
```

Block Patterns

should_load_remote_block_patterns

Default `true`. The filter is checked when registering remote block patterns, set to `false` to disable.

For example, to disable use:

```
add_filter( 'should_load_remote_block_patterns', '__return_false' );
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Editor Hooks](#)

[Previous Block Filters](#) [Previous: Block Filters](#)

[Next i18n Filters](#) [Next: i18n Filters](#)

i18n Filters

In this article

Table of Contents

- [Filter Arguments](#)
 - [i18n.gettext](#)
 - [i18n.gettext_with_context](#)
 - [i18n.nggettext](#)
 - [i18n.nggettext_with_context](#)
- [Basic Example](#)
- [Using ‘text domain’-specific filters](#)

[↑ Back to top](#)

The i18n functions (`__()`, `_x()`, `_n()` and `_nx()`) provide translations of strings for use in your code. The values returned by these functions are filterable if you need to override them, using the following filters:

- [i18n.gettext](#)
- [i18n.gettext_with_context](#)
- [i18n.nggettext](#)
- [i18n.nggettext_with_context](#)

[Filter Arguments](#)

The filters are passed the following arguments, in line with their PHP equivalents.

[i18n.gettext](#)

```
function i18nGettextCallback( translation, text, domain ) {  
    return translation;  
}
```

i18n.gettext with context

```
function i18nGettextWithContextCallback( translation, text, context, domain ) {
    return translation;
}
```

i18n.ngettext

```
function i18nNgettextCallback( translation, single, plural, number, domain ) {
    return translation;
}
```

i18n.ngettext with context

```
function i18nNgettextWithContextCallback(
    translation,
    single,
    plural,
    number,
    context,
    domain
) {
    return translation;
}
```

Basic Example

Here is a simple example, using the `i18n.gettext` filter to override a specific translation.

```
// Define our filter callback.
function myPluginGettextFilter( translation, text, domain ) {
    if ( text === 'Create Reusable block' ) {
        return 'Save to MyOrg block library';
    }
    return translation;
}

// Adding the filter
wp.hooks.addFilter(
    'i18n gettext',
    'my-plugin/override-add-to-reusable-blocks-label',
    myPluginGettextFilter
);
```

Using ‘text domain’-specific filters

Filters that are specific to the text domain you’re operating on are generally preferred for performance reasons (since your callback will only be run for strings in the relevant text domain).

To attach to a text domain-specific filter append an underscore and the text-domain to the standard filter name. For example, if filtering a string where the text domain is “woocommerce”, you would use one of the following filters:

- `i18ngettext_woocommerce`
- `i18ngettext_with_context_woocommerce`
- `i18ngettext_woocommerce`
- `i18ngettext_with_context_woocommerce`

For example:

```
// Define our filter callback.
function myPluginGettextFilter( translation, text, domain ) {
    if ( text === 'You've fulfilled all your orders' ) {
        return 'All packed up and ready to go. Good job!';
    }
    return translation;
}

// Adding the filter
wp.hooks.addFilter(
    'i18ngettext_woocommerce',
    'my-plugin/override-fulfilled-all-orders-text',
    myPluginGettextFilter
);
```

Note: To apply a filter where the text-domain is `undefined` (for example WordPress core strings), then use the name “`default`” to construct the filter name.

- `i18ngettext_default`
- `i18ngettext_with_context_default`
- `i18ngettext_default`
- `i18ngettext_with_context_default`

First published

July 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: i18n Filters](#)

[Previous Editor Hooks](#) [Previous: Editor Hooks](#)
[Next Parser Filters](#) [Next: Parser Filters](#)

Parser Filters

In this article

Table of Contents

- [Server-side parser](#)
- [Client-side parser](#)
- [Filters](#)

[↑ Back to top](#)

When the editor is interacting with blocks, these are stored in memory as data structures comprising a few basic properties and attributes. Upon saving a working post we serialize these data structures into a specific HTML structure and save the resultant string into the `post_content` property of the post in the WordPress database. When we load that post back into the editor we have to make the reverse transformation to build those data structures from the serialized format in HTML.

The process of loading the serialized HTML into the editor is performed by the *block parser*. The formal specification for this transformation is encoded in the parsing expression grammar (PEG) inside the `@wordpress/block-serialization-spec-parser` package. The editor provides a default parser implementation of this grammar but there may be various reasons for replacing that implementation with a custom implementation. We can inject our own custom parser implementation through the appropriate filter.

[**Server-side parser**](#)

Plugins have access to the parser if they want to process posts in their structured form instead of a plain HTML-as-string representation.

[**Client-side parser**](#)

The editor uses the client-side parser while interactively working in a post. The plain HTML-as-string representation is sent to the browser by the backend and then the editor performs the first parse to initialize itself.

[**Filters**](#)

To replace the server-side parser, use the `block_parser_class` filter. The filter transforms the string class name of a parser class. This class is expected to expose a `parse` method.

Example:

```
class EmptyParser {  
    public function parse( $post_content ) {  
        // return an empty document  
        return array();  
    }  
}
```

```
}

function wpdocs_select_empty_parser( $prev_parser_class ) {
    return 'EmptyParser';
}

add_filter( 'block_parser_class', 'wpdocs_select_empty_parser', 10, 1 );
```

Note: At the present time it's not possible to replace the client-side parser.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Parser Filters”](#)

[Previous i18n Filters](#) [Previous: i18n Filters](#)
[Next Autocomplete](#) [Next: Autocomplete](#)

Autocomplete

[↑ Back to top](#)

The `editor.Autocomplete.completers` filter is for extending and overriding the list of autocompleters used by blocks.

The `Autocomplete` component found in `@wordpress/block-editor` applies this filter. The `@wordpress/components` package provides the foundational `Autocomplete` component that does not apply such a filter, but blocks should generally use the component provided by `@wordpress/block-editor`.

Example

Here is an example of using the `editor.Autocomplete.completers` filter to add an acronym completer. You can find full documentation for the autocomplete interface with the `Autocomplete` component in the `@wordpress/components` package.

```
// Our completer
const acronymCompleter = {
    name: 'acronyms',
    triggerPrefix: ':::',
    options: [
        { letters: 'FYI', expansion: 'For Your Information' },
        { letters: 'AFAIK', expansion: 'As Far As I Know' },
        { letters: 'IIRC', expansion: 'If I Recall Correctly' },
    ]
}
```

```

        ],
        getOptionKeywords( { letters, expansion } ) {
            const expansionWords = expansion.split( /\s+/ );
            return [ letters, ...expansionWords ];
        },
        getOptionLabel: acronym => acronym.letters,
        getOptionCompletion: ( { letters, expansion } ) => (
            <abbr title={ expansion }>{ letters }</abbr>,
        ),
    );
}

// Our filter function
function appendAcronymCompleter( completers, blockName ) {
    return blockName === 'my-plugin/foo' ?
        [ ...completers, acronymCompleter ] :
        completers;
}

// Adding the filter
wp.hooks.addFilter(
    'editor.Autocomplete.completers',
    'my-plugin/autocomplete/acronym',
    appendAcronymCompleter
);

```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Autocomplete”](#)

[Previous Parser Filters](#) [Previous: Parser Filters](#)
[Next Global Styles Filters](#) [Next: Global Styles Filters](#)

Global Styles Filters

[↑ Back to top](#)

WordPress 6.1 has introduced some server-side filters to hook into the `theme.json` data provided at the different data layers:

- `wp_theme_json_data_default`: hooks into the default data provided by WordPress
- `wp_theme_json_data_blocks`: hooks into the data provided by the blocks
- `wp_theme_json_data_theme`: hooks into the data provided by the theme
- `wp_theme_json_data_user`: hooks into the data provided by the user

Each filter receives an instance of the `WP_Theme_JSON_Data` class with the data for the respective layer. To provide new data, the filter callback needs to use the `update_with($new_data)` method, where `$new_data` is a valid `theme.json`-like structure. As with any `theme.json`, the new data needs to declare which `version` of the `theme.json` is using, so it can correctly be migrated to the runtime one, should it be different.

Example:

This is how to pass a new color palette for the theme and disable the text color UI:

```
function wpdocs_filter_theme_json_theme( $theme_json ){
    $new_data = array(
        'version' => 2,
        'settings' => array(
            'color' => array(
                'text' => false,
                'palette' => array( /* New palette */
                    array(
                        'slug' => 'foreground',
                        'color' => 'black',
                        'name' => __( 'Foreground', 'theme-domain' ),
                    ),
                    array(
                        'slug' => 'background',
                        'color' => 'white',
                        'name' => __( 'Background', 'theme-domain' ),
                    ),
                ),
            ),
        );
    );

    return $theme_json->update_with( $new_data );
}
add_filter( 'wp_theme_json_data_theme', 'wpdocs_filter_theme_json_theme' )
```

First published

September 13, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Global Styles Filters](#)

[Previous Autocomplete](#) [Previous: Autocomplete](#)
[Next SlotFills](#) [Reference Next: SlotFills](#) [Reference](#)

SlotFills Reference

In this article

Table of Contents

- [Usage overview](#)
- [How do they work?](#)
- [Currently available SlotFills and examples](#)

[↑ Back to top](#)

Slot and Fill are components that have been exposed to allow developers to inject items into some predefined places in the Gutenberg admin experience.

Please see the [SlotFill component docs](#) for more details.

In order to use them, we must leverage the [@wordpress/plugins](#) api to register a plugin that will inject our items.

[Usage overview](#)

In order to access the SlotFills, we need to do four things:

1. Import the `registerPlugin` method from `wp.plugins`.
2. Import the SlotFill we want from `wp.editPost`.
3. Define a method to render our changes. Our changes/additions will be wrapped in the SlotFill component we imported.
4. Register the plugin.

Here is an example using the `PluginPostStatusInfo` slotFill:

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginPostStatusInfo } from '@wordpress/edit-post';

const PluginPostStatusInfoTest = () => (
    <PluginPostStatusInfo>
        <p>Post Status Info SlotFill</p>
    </PluginPostStatusInfo>
);

registerPlugin( 'post-status-info-test', { render: PluginPostStatusInfoTes
```

[How do they work?](#)

SlotFills are created using `createSlotFill`. This creates two components, `Slot` and `Fill` which are then used to create a new component that is exported on the `wp.plugins` global.

Definition of the `PluginPostStatusInfo` SlotFill ([see core code](#))

```

/**
 * Defines as extensibility slot for the Summary panel.
 */

/**
 * WordPress dependencies
 */
import { createSlotFill, PanelRow } from '@wordpress/components';

export const { Fill, Slot } = createSlotFill( 'PluginPostStatusInfo' );

const PluginPostStatusInfo = ( { children, className } ) => (
    <Fill>
        <PanelRow className={ className }>{ children }</PanelRow>
    </Fill>
);

PluginPostStatusInfo.Slot = Slot;

export default PluginPostStatusInfo;

```

This new Slot is then exposed in the editor. The example below is from core and represents the Summary panel.

As we can see, the `<PluginPostStatusInfo.Slot>` is wrapping all of the items that will appear in the panel.

Any items that have been added via the `SlotFill` (see the example above), will be included in the `fills` parameter and be displayed between the `<PostAuthor/>` and `<PostTrash/>` components.

See [core code](#).

```

const PostStatus = ( { isOpened, onTogglePanel } ) => (
    <PanelBody
        className="edit-post-post-status"
        title={ __( 'Summary' ) }
        opened={ isOpened }
        onToggle={ onTogglePanel }
    >
        <PluginPostStatusInfo.Slot>
            { ( fills ) => (
                <>
                    <PostVisibility />
                    <PostSchedule />
                    <PostFormat />
                    <PostSticky />
                    <PostPendingStatus />
                    <PostAuthor />
                    { fills }
                    <PostTrash />
                </>
            ) }
    </PluginPostStatusInfo.Slot>

```

```
</PanelBody>
);
```

Currently available SlotFills and examples

The following SlotFills are available in the edit-post package. Please refer to the individual items below for usage and example details:

- [MainDashboardButton](#)
- [PluginBlockSettingsMenuItem](#)
- [PluginDocumentSettingPanel](#)
- [PluginMoreMenuItem](#)
- [PluginPostPublishPanel](#)
- [PluginPostStatusInfo](#)
- [PluginPrePublishPanel](#)
- [PluginSidebar](#)
- [PluginSidebarMoreMenuItem](#)

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: SlotFills Reference](#)”

[Previous Global Styles Filters Previous: Global Styles Filters](#)
[Next MainDashboardButton Next: MainDashboardButton](#)

MainDashboardButton

In this article

Table of Contents

- [Examples](#)
 - [Post editor example](#)
 - [Site editor example](#)

[↑ Back to top](#)

This slot allows replacing the default main dashboard button in the post editor and site editor. It's used for returning back to main wp-admin dashboard when editor is in fullscreen mode.

Examples

Post editor example

This will override the W icon button in the header.

```
import { registerPlugin } from '@wordpress/plugins';
import { __experimentalMainDashboardButton as MainDashboardButton } from 'wp-element';

const MainDashboardButtonTest = () => (
    <MainDashboardButton>
        Custom main dashboard button content
    </MainDashboardButton>
);

registerPlugin( 'main-dashboard-button-test', {
    render: MainDashboardButtonTest,
} );
```

If your goal is just to replace the icon of the existing button in the post editor, that can be achieved in the following way:

```
import { registerPlugin } from '@wordpress/plugins';
import {
    __experimentalFullscreenModeClose as FullscreenModeClose,
    __experimentalMainDashboardButton as MainDashboardButton,
} from '@wordpress/edit-post';
import { close } from '@wordpress/icons';

const MainDashboardButtonIconTest = () => (
    <MainDashboardButton>
        <FullscreenModeClose icon={ close } href="http://wordpress.org" />
    </MainDashboardButton>
);

registerPlugin( 'main-dashboard-button-icon-test', {
    render: MainDashboardButtonIconTest,
} );
```

Site editor example

In the site editor this slot refers to the “back to dashboard” button in the navigation sidebar.

```
import { registerPlugin } from '@wordpress/plugins';
import { __experimentalMainDashboardButton as MainDashboardButton } from 'wp-element';
import { __experimentalNavigationBackButton as NavigationBackButton } from 'wp-components';

const MainDashboardButtonIconTest = () => (
    <MainDashboardButton>
        <NavigationBackButton
            backButtonLabel={ __( 'Back to dashboard' ) }
            className="edit-site-navigation-panel__back-to-dashboard"
            href="index.php"
        >
```

```
        />
    </MainDashboardButton>
);

registerPlugin( 'main-dashboard-button-icon-test', {
    render: MainDashboardButtonIconTest,
} );
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: MainDashboardButton](#)

[Previous SlotFills Reference](#) [Previous: SlotFills Reference](#)

[Next PluginBlockSettingsMenuItem](#) [Next: PluginBlockSettingsMenuItem](#)

PluginBlockSettingsMenuItem

In this article

Table of Contents

- [Example](#)
- [Location](#)

[↑ Back to top](#)

This slot allows for adding a new item into the More Options area.

This will either appear in the controls for each block or at the Top Toolbar depending on the users setting.

[Example](#)

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginBlockSettingsMenuItem } from '@wordpress/edit-post';

const PluginBlockSettingsMenuGroupTest = () => (
    <PluginBlockSettingsMenuItem
        allowedBlocks={ [ 'core/paragraph' ] }
        icon="smiley"
        label="Menu item text"
        onClick={ () => {
            alert( 'clicked' );
        } }
    >
```

```
/>  
);  
  
registerPlugin( 'block-settings-menu-group-test', {  
    render: PluginBlockSettingsMenuGroupTest,  
} );
```

Location



First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: PluginBlockSettingsMenuItem](#)

[Previous MainDashboardButton](#) [Previous: MainDashboardButton](#)
[Next PluginDocumentSettingPanel](#) [Next: PluginDocumentSettingPanel](#)

PluginDocumentSettingPanel

In this article

Table of Contents

- [Available Props](#)
- [Example](#)
- [Accessing a panel programmatically](#)

[↑ Back to top](#)

This SlotFill allows registering a UI to edit Document settings.

[Available Props](#)

- **name** string: A string identifying the panel.
- **className** string: An optional class name added to the sidebar body.
- **title** string: Title displayed at the top of the sidebar.
- **icon** (string|Element): The [Dashicon](#) icon slug string, or an SVG WP element.

[Example](#)

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginDocumentSettingPanel } from '@wordpress/edit-post';

const PluginDocumentSettingPanelDemo = () => (
    <PluginDocumentSettingPanel
        name="custom-panel"
        title="Custom Panel"
        className="custom-panel"
    >
        Custom Panel Contents
    </PluginDocumentSettingPanel>
);

registerPlugin( 'plugin-document-setting-panel-demo' , {
    render: PluginDocumentSettingPanelDemo,
    icon: 'palmtree',
} );
```

[Accessing a panel programmatically](#)

Core and custom panels can be access programmatically using their panel name. The core panel names are:

- Summary Panel: `post-status`
- Categories Panel: `taxonomy-panel-category`
- Tags Panel: `taxonomy-panel-post_tag`

- Featured Image Panel: `featured-image`
- Excerpt Panel: `post-excerpt`
- DiscussionPanel: `discussion-panel`

Custom panels are namespaced with the plugin name that was passed to `registerPlugin`. In order to access the panels using function such as `toggleEditorPanelOpened` or `toggleEditorPanelEnabled` be sure to prepend the namespace.

To programmatically toggle panels, use the following:

```
import { useDispatch } from '@wordpress/data';
import { store as editorStore } from '@wordpress/editor';

const Example = () => {
    const { toggleEditorPanelOpened } = useDispatch( editorStore );
    return (
        <Button
            variant="primary"
            onClick={ () => {
                // Toggle the Summary panel
                toggleEditorPanelOpened( 'post-status' );

                // Toggle the Custom Panel introduced in the example above
                toggleEditorPanelOpened(
                    'plugin-document-setting-panel-demo/custom-panel'
                );
            } }
        >
            Toggle Panels
        </Button>
    );
};


```

It is also possible to remove panels from the admin using the `removeEditorPanel` function by passing the name of the registered panel.

```
import { useDispatch } from '@wordpress/data';
import { store as editorStore } from '@wordpress/editor';

const Example = () => {
    const { removeEditorPanel } = useDispatch( editorStore );
    return (
        <Button
            variant="primary"
            onClick={ () => {
                // Remove the Featured Image panel.
                removeEditorPanel( 'featured-image' );

                // Remove the Custom Panel introduced in the example above
                removeEditorPanel(
                    'plugin-document-setting-panel-demo/custom-panel'
                );
            } }
        >
    );
};


```

```
        Toggle Panels
    </Button>
);
};
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: PluginDocumentSettingPanel](#)"

[Previous PluginBlockSettingsMenuItem](#) [Previous: PluginBlockSettingsMenuItem](#)
[Next PluginMoreMenuItem](#) [Next: PluginMoreMenuItem](#)

PluginMoreMenuItem

In this article

Table of Contents

- [Example](#)
- [Location](#)

[↑ Back to top](#)

This slot will add a new item to the More Tools & Options section.

[Example](#)

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginMoreMenuItem } from '@wordpress/edit-post';
import { image } from '@wordpress/icons';

const MyButtonMoreMenuItemTest = () => (
    <PluginMoreMenuItem
        icon={ image }
        onClick={ () => {
            alert( 'Button Clicked' );
        } }
    >
        More Menu Item
    </PluginMoreMenuItem>
);

registerPlugin( 'more-menu-item-test', { render: MyButtonMoreMenuItemTest }
```

Location

The screenshot shows the Craft CMS Document Settings panel. At the top, there are buttons for 'Saved' (with a checkmark), 'Preview', 'Publish...', a gear icon for settings, and a three-dot menu icon. Below this is a 'View' section with a 'Top Toolbar' option checked, which allows access to all block and document tools in one place. There are also 'Spotlight Mode' and 'Fullscreen Mode' options. The 'Editor' section has 'Visual Editor' checked. The 'Plugins' section contains a 'More Menu Item' option, which is highlighted with a blue border. The 'Tools' section includes 'Manage All Reusable Blocks', 'Keyboard Shortcuts' (with a keyboard icon), and 'Copy All Content'. The 'Options' section is at the bottom.

✓ Saved Preview Publish... ⚙ :

View

Top Toolbar
✓ Access all block and document tools in a single place

Spotlight Mode
Focus on one block at a time

Fullscreen Mode
Work without distraction

Editor

✓ Visual Editor

Code Editor ☰ ⌘M

Plugins

More Menu Item

Tools

Manage All Reusable Blocks

Keyboard Shortcuts ^⌘H

Copy All Content

Options

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: PluginMoreMenuItem](#)

[Previous PluginDocumentSettingPanel](#) [Previous: PluginDocumentSettingPanel](#)

[Next PluginPostPublishPanel](#) [Next: PluginPostPublishPanel](#)

PluginPostPublishPanel

In this article

Table of Contents

- [Example](#)
- [Location](#)

[↑ Back to top](#)

This slot allows for injecting items into the bottom of the post-publish panel that appears after a post is published.

[Example](#)

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginPostPublishPanel } from '@wordpress/edit-post';

const PluginPostPublishPanelTest = () => (
    <PluginPostPublishPanel>
        <p>Post Publish Panel</p>
    </PluginPostPublishPanel>
);

registerPlugin( 'post-publish-panel-test', {
    render: PluginPostPublishPanelTest,
} );
```

Location

Published X

[SlotFills](#) is now live.

What's next?

Post address

<http://rac.test/slotfills/>

[View Post](#) [Copy Link](#)

Post Publish Panel

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: PluginPostPublishPanel](#)

[Previous PluginMoreMenuItem](#) [Previous: PluginMoreMenuItem](#)
[Next PluginPostStatusInfo](#) [Next: PluginPostStatusInfo](#)

PluginPostStatusInfo

In this article

[Table of Contents](#)

- [Example](#)
- [Location](#)

[↑ Back to top](#)

This slot allows for the insertion of items in the Summary panel of the document sidebar.

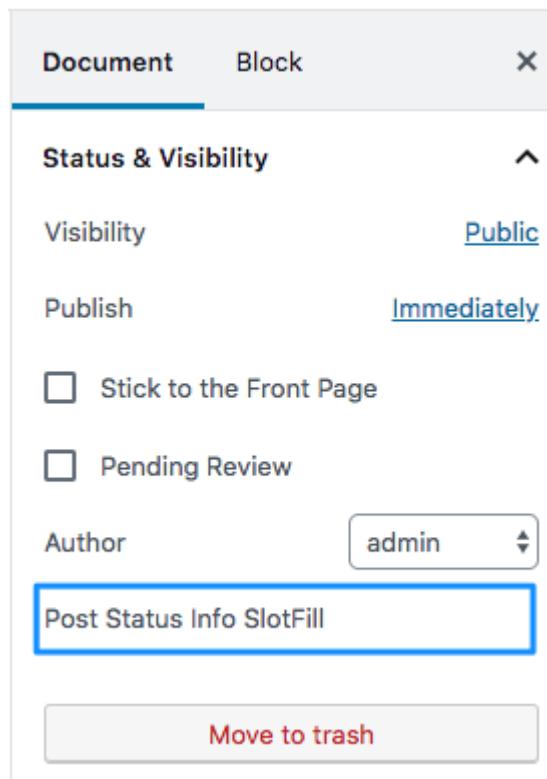
Example

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginPostStatusInfo } from '@wordpress/edit-post';

const PluginPostStatusInfoTest = () => (
  <PluginPostStatusInfo>
    <p>Post Status Info SlotFill</p>
  </PluginPostStatusInfo>
);

registerPlugin( 'post-status-info-test', { render: PluginPostStatusInfoTest } );
```

Location



First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: PluginPostStatusInfo](#)

[Previous PluginPostPublishPanel](#) [Previous: PluginPostPublishPanel](#)

[Next PluginPrePublishPanel](#) [Next: PluginPrePublishPanel](#)

PluginPrePublishPanel

In this article

Table of Contents

- [Example](#)
- [Location](#)

[↑ Back to top](#)

This slot allows for injecting items into the bottom of the pre-publish panel that appears to confirm publishing after the user clicks “Publish”.

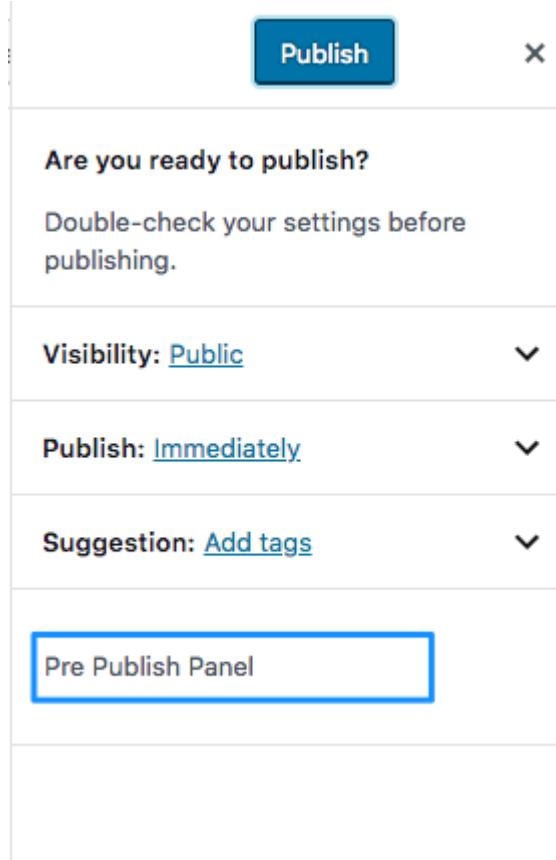
[Example](#)

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginPrePublishPanel } from '@wordpress/edit-post';

const PluginPrePublishPanelTest = () => (
    <PluginPrePublishPanel>
        <p>Pre Publish Panel</p>
    </PluginPrePublishPanel>
);

registerPlugin( 'pre-publish-panel-test', {
    render: PluginPrePublishPanelTest,
} );
```

Location



First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: PluginPrePublishPanel](#)

[Previous PluginPostStatusInfo](#) [Previous: PluginPostStatusInfo](#)
[Next PluginSidebar](#) [Next: PluginSidebar](#)

PluginSidebar

In this article

Table of Contents

- [Example](#)
- [Location](#)
 - [Closed State](#)
 - [Open State](#)

[↑ Back to top](#)

This slot allows for adding items into the Gutenberg Toolbar.

Using this slot will add an icon to the bar that, when clicked, will open a sidebar with the content of the items wrapped in the `<PluginSidebar />` component.

Example

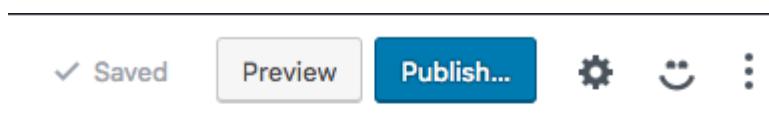
```
import { registerPlugin } from '@wordpress/plugins';
import { PluginSidebar } from '@wordpress/edit-post';
import { image } from '@wordpress/icons';

const PluginSidebarTest = () => (
  <PluginSidebar name="plugin-sidebar-test" title="My Plugin" icon={ image( 48, 48, { alt: 'My Plugin' } ) }>
    <p>Plugin Sidebar</p>
  </PluginSidebar>
);

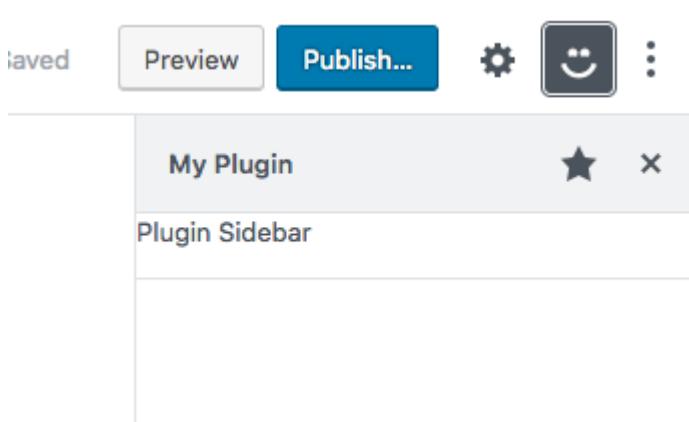
registerPlugin( 'plugin-sidebar-test', { render: PluginSidebarTest } );
```

Location

Closed State



Open State



First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: PluginSidebar](#)

[Previous PluginPrePublishPanel](#) [Previous: PluginPrePublishPanel](#)

[Next PluginSidebarMoreMenuItem](#) [Next: PluginSidebarMoreMenuItem](#)

PluginSidebarMoreMenuItem

In this article

Table of Contents

- [Example](#)
- [Location](#)

[↑ Back to top](#)

This slot allows the creation of a <PluginSidebar> with a menu item that when clicked will expand the sidebar to the appropriate Plugin section.

This is done by setting the target on <PluginSidebarMoreMenuItem> to match the name on the <PluginSidebar>

[Example](#)

```
import { registerPlugin } from '@wordpress/plugins';
import { PluginSidebar, PluginSidebarMoreMenuItem } from '@wordpress/edit-
import { image } from '@wordpress/icons';

const PluginSidebarMoreMenuItemTest = () => (
  <>
    <PluginSidebarMoreMenuItem target="sidebar-name" icon={ image }>
      Expanded Sidebar - More item
    </PluginSidebarMoreMenuItem>
    <PluginSidebar name="sidebar-name" icon={ image } title="My Sidebar">
      Content of the sidebar
    </PluginSidebar>
  </>
);
registerPlugin( 'plugin-sidebar-expanded-test', {
  render: PluginSidebarMoreMenuItemTest,
} );
```

Location



First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: PluginSidebarMoreMenuItem”](#)

[Previous PluginSidebar](#) [Previous: PluginSidebar](#)
[Next RichText Reference](#) [Next: RichText Reference](#)

RichText Reference

In this article

Table of Contents

- [Property reference](#)
- [Core blocks using the RichText component](#)
- [Example](#)
- [Common issues and solutions](#)
 - [HTML formatting tags display in the content](#)
 - [Unwanted formatting options still display](#)
 - [Disable specific format types in Editor](#)

[↑ Back to top](#)

RichText is a component that allows developers to render a [contenteditable input](#), providing users with the option to format block content to make it bold, italics, linked, or use other formatting.

The RichText component is extremely powerful because it provides built-in functionality you won't find in other components:

- **Consistent Styling in the Admin and Frontend:** The editable container can be set to any block-level element, such as a `div`, `h2` or `p` tag. This allows the styles you apply in `style.css` to more easily apply on the frontend and admin, without having to rewrite them in `editor.css`.
- **Cohesive Placeholder Text:** Before the user writes their content, it's easy to include placeholder text that's already styled to match the rest of the block editor.
- **Control Over Formatting Options:** It's possible to dictate exactly which formatting options you want to allow for the RichText field. For example, you can dictate whether to allow the user to make text bold, italics or both.

Unlike other components that exist in the [Component Reference](#) section, RichText lives separately because it only makes sense within the block editor, and not within other areas of WordPress.

[Property reference](#)

For a list of the possible properties to pass your RichText component, [check out the component documentation on GitHub](#).

[Core blocks using the RichText component](#)

There are a number of core blocks using the RichText component. The JavaScript edit function linked below for each block can be used as a best practice reference while creating your own blocks.

- [Button](#): RichText is used to enter the button's text.
- [Heading](#): RichText is used to enter the heading's text.

- **Quote:** RichText is used in two places, for both the quotation and citation text.
- **Search:** RichText is used in two places, for both the label above the search field and the submit button text.

Example

```
import { registerBlockType } from '@wordpress/blocks';
import { useBlockProps, RichText } from '@wordpress/block-editor';

registerBlockType( /* ... */ , {
    // ...

    attributes: {
        content: {
            type: 'string',
            source: 'html',
            selector: 'h2',
        },
    },
    edit( { attributes, setAttributes } ) {
        const blockProps = useBlockProps();

        return (
            <RichText
                { ...blockProps }
                tagName="h2" // The tag here is the element output and edi
                value={ attributes.content } // Any existing content, eith
                allowedFormats={ [ 'core/bold', 'core/italic' ] } // Allow
                onChange={ ( content ) => setAttributes( { content } ) } / /
                placeholder={ __( 'Heading...' ) } // Display this text be
            />
        );
    },
    save( { attributes } ) {
        const blockProps = useBlockProps.save();

        return <RichText.Content { ...blockProps } tagName="h2" value={ at
    }
} );
```

Common issues and solutions

While using the RichText component a number of common issues tend to appear.

HTML formatting tags display in the content

If the HTML tags from text formatting such as `` or `` are being escaped and displayed on the frontend of the site, this is likely due to an issue in your save function. Make sure your code looks something like `<RichText.Content tagName="h2" value={`

```
heading } /> (JSX) within your save function instead of simply outputting the value with  
<h2>{ heading }</h2>.
```

[Unwanted formatting options still display](#)

Before moving forward, consider if using the `RichText` component makes sense at all. Would it be better to use a basic `input` or `textarea` element? If you don't think any formatting should be possible, these HTML tags may make more sense.

If you'd still like to use `RichText`, you can eliminate all of the formatting options by specifying the `withoutInteractiveFormatting` property.

If you want to limit the formats allowed, you can specify using `allowedFormats` property in your code, see the example above or [the component documentation](#) for details.

[Disable specific format types in Editor](#)

The `RichText` component uses formats to display inline elements, for example images within the paragraph block. If you just want to disable a format from the editor, you can use the `unregisterFormatType` function. For example to disable inline images, use:

```
wp.richText.unregisterFormatType( 'core/image' );
```

To apply, you would need to enqueue the above script in your plugin or theme. See the JavaScript tutorial for [how to load JavaScript in WordPress](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: RichText Reference](#)

[Previous PluginSidebarMoreMenuItem](#) [Previous: PluginSidebarMoreMenuItem](#)
[Next Theme.json Reference](#) [Next: Theme.json Reference](#)

Theme.json Reference

[↑ Back to top](#)

This reference guide lists the settings and style properties defined in the `theme.json` schema. See the [theme.json how to guide](#) for examples and guide on how to use the `theme.json` file in your theme.

- [Version 2 \(living reference\)](#)

Older versions

- [Migrating to Newer Theme.json Versions](#)
- [Version 1](#)

First published

January 4, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Theme.json Reference”](#)

[Previous RichText Reference](#) [Previous: RichText Reference](#)
[Next Theme.json Version 2](#) [Next: Theme.json Version 2](#)

Theme.json Version 2

In this article

Table of Contents

- [Schema](#)
- [Settings](#)
 - [appearanceTools](#)
 - [useRootPaddingAwareAlignments](#)
 - [border](#)
 - [shadow](#)
 - [color](#)
 - [background](#)
 - [dimensions](#)
 - [layout](#)
 - [lightbox](#)
 - [position](#)
 - [spacing](#)
 - [typography](#)
 - [custom](#)
- [Styles](#)
 - [border](#)
 - [color](#)
 - [dimensions](#)
 - [spacing](#)
 - [typography](#)
 - [filter](#)
 - [shadow](#)
 - [outline](#)
 - [css](#)
- [customTemplates](#)

- [templateParts](#)
- [Patterns](#)

[↑ Back to top](#)

This is the living specification for **version 2** of theme.json. This version works with WordPress 5.9 or later, and the latest Gutenberg plugin.

There are some related documents that you may be interested in:

- the [theme.json v1](#) specification, and
- the [reference to migrate from theme.json v1 to v2](#).

This reference guide lists the settings and style properties defined in the theme.json schema. See the [theme.json how to guide](#) for examples and guidance on how to use the theme.json file in your theme.

Schema

Remembering the theme.json settings and properties while you develop can be difficult, so a [JSON schema](#) was created to help.

Code editors can pick up the schema and can provide helpful hints and suggestions such as tooltips, autocomplete, or schema validation in the editor. To use the schema in Visual Studio Code, add `$schema: "https://schemas.wp.org/trunk/theme.json"` to the beginning of your theme.json file together with a `version` corresponding to the version you wish to use, e.g.:

```
{  
  "$schema": "https://schemas.wp.org/trunk/theme.json",  
  "version": 2,  
  ...  
}
```

Settings

appearanceTools

Setting that enables the following UI tools:

- background: backgroundImage
- border: color, radius, style, width
- color: link
- dimensions: aspectRatio, minHeight
- position: sticky
- spacing: blockGap, margin, padding
- typography: lineHeight

useRootPaddingAwareAlignments

Note: Since WordPress 6.1.

Enables root padding (the values from `styles.spacing.padding`) to be applied to the contents of full-width blocks instead of the root block.

Please note that when using this setting, `styles.spacing.padding` should always be set as an object with `top`, `right`, `bottom`, `left` values declared separately.

[border](#)

Settings related to borders.

Property	Type	Default Props
-----------------	-------------	----------------------

color	boolean	false
radius	boolean	false
style	boolean	false
width	boolean	false

[shadow](#)

Settings related to shadows.

Property	Type	Default Props
-----------------	-------------	----------------------

defaultPresets	boolean	true
presets	array	name, shadow, slug

[color](#)

Settings related to colors.

Property	Type	Default Props
-----------------	-------------	----------------------

background	boolean	true
custom	boolean	true
customDuotone	boolean	true
customGradient	boolean	true
defaultDuotone	boolean	true
defaultGradients	boolean	true
defaultPalette	boolean	true
duotone	array	colors, name, slug
gradients	array	gradient, name, slug
link	boolean	false
palette	array	color, name, slug
text	boolean	true
heading	boolean	true
button	boolean	true

[background](#)

Settings related to background.

Property	Type	Default Props
backgroundImage	boolean	false

[dimensions](#)

Settings related to dimensions.

Property	Type	Default Props
aspectRatio	boolean	false
minHeight	boolean	false

[layout](#)

Settings related to layout.

Property	Type	Default Props
contentSize	string	
wideSize	string	
allowEditing	boolean	true
allowCustomContentAndWideSize	boolean	true

[lightbox](#)

Settings related to the lightbox.

Property	Type	Default Props
enabled	boolean	
allowEditing	boolean	

[position](#)

Settings related to position.

Property	Type	Default Props
sticky	boolean	false

[spacing](#)

Settings related to spacing.

Property	Type	Default	Props
blockGap	undefined	null	
margin	boolean	false	
padding	boolean	false	
units	array	px,em,rem,vh,vw,%	
customSpacingSize	boolean	true	
spacingSizes	array		name, size, slug
spacingScale	object		

[typography](#)

Settings related to typography.

Property	Type	Default	Props
defaultFontSizes	boolean	true	
customFontSize	boolean	true	
fontStyle	boolean	true	
fontWeight	boolean	true	
fluid	undefined	false	
letterSpacing	boolean	true	
lineHeight	boolean	false	
textColumns	boolean	false	
textDecoration	boolean	true	
writingMode	boolean	false	
textTransform	boolean	true	
dropCap	boolean	true	
fontSizes	array		fluid, name, size, slug
fontFamilies	array		fontFace, fontFamily, name, slug

[custom](#)

Generate custom CSS custom properties of the form `--wp--custom--{key}--{nested-key}: {value};`. camelCased keys are transformed to kebab-case as to follow the CSS property naming schema. Keys at different depth levels are separated by `--`, so keys should not include `--` in the name.

[Styles](#)

[border](#)

Border styles.

Property	Type	Props
color	string	object

Property	Type	Props
radius	string, object	
style	string, object	
width	string, object	
top	object	color, style, width
right	object	color, style, width
bottom	object	color, style, width
left	object	color, style, width

color

Color styles.

Property	Type	Props
background	string, object	
gradient	string, object	
text	string, object	

dimensions

Dimensions styles

Property	Type	Props
aspectRatio	string, object	
minHeight	string, object	

spacing

Spacing styles.

Property	Type	Props
blockGap	string, object	
margin	object	bottom, left, right, top
padding	object	bottom, left, right, top

typography

Typography styles.

Property	Type	Props
fontFamily	string, object	
fontSize	string, object	
fontStyle	string, object	
fontWeight	string, object	

Property	Type	Props
letterSpacing	string, object	
lineHeight	string, object	
textColumns	string	
textDecoration	string, object	
writingMode	string, object	
textTransform	string, object	

filter

CSS and SVG filter styles.

Property	Type	Props
duotone	string, object	

shadow

Box shadow styles.

outline

Outline styles.

Property	Type	Props
color	string, object	
offset	string, object	
style	string, object	
width	string, object	

css

Sets custom CSS to apply styling not covered by other theme.json properties.

customTemplates

Additional metadata for custom templates defined in the templates folder.

Type: object.

Property	Description	Type
name	Filename, without extension, of the template in the templates folder.	string
title	Title of the template, translatable.	string

Property Description	Type
postTypes List of post types that can use this custom template.	array

[templateParts](#)

Additional metadata for template parts defined in the parts folder.

Type: object.

Property Description	Type
name Filename, without extension, of the template in the parts folder.	string
title Title of the template, translatable.	string
area The area the template part is used for. Block variations for header and footer values exist and will be used when the area is set to one of those.	string

[Patterns](#)

An array of pattern slugs to be registered from the Pattern Directory.

Type: array.

First published

January 14, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Theme.json Version 2"](#)

[Previous Theme.json Reference](#) [Previous: Theme.json Reference](#)

[Next Theme.json Version 1 Reference](#) [Next: Theme.json Version 1 Reference](#)

Theme.json Version 1 Reference

In this article

Table of Contents

- [Settings](#)
 - [border](#)
 - [color](#)
 - [layout](#)
 - [spacing](#)
 - [typography](#)
 - [custom](#)

- [Styles](#)
 - [border](#)
 - [color](#)
 - [spacing](#)
 - [typography](#)

[↑ Back to top](#)

Theme.json version 2 has been released, see the [theme.json migration guide](#) for updating to the latest version.

[Settings](#)

[border](#)

Settings related to borders.

Property	Type	Default Props
customRadius	boolean	false

[color](#)

Settings related to colors.

Property	Type	Default Props
custom	boolean	true
customDuotone	boolean	true
customGradient	boolean	true
duotone	array	colors, name, slug
gradients	array	gradient, name, slug
link	boolean	false
palette	array	color, name, slug

[layout](#)

Settings related to layout.

Property	Type	Default Props
contentSize	string	
wideSize	string	

[spacing](#)

Settings related to spacing.

Property	Type	Default	Props
customMargin	boolean	false	
customPadding	boolean	false	
units	array	px,em,rem,vh,vw,%	

[typography](#)

Settings related to typography.

Property	Type	Default	Props
customFontSize	boolean	true	
customLineHeight	boolean	false	
dropCap	boolean	true	
fontSizes	array	name, size, slug	

[custom](#)

Generate custom CSS custom properties of the form `--wp--custom--{key}--{nested-key}: {value};`. camelCased keys are transformed to kebab-case as to follow the CSS property naming schema. Keys at different depth levels are separated by `--`, so keys should not include `--` in the name.

[Styles](#)

[border](#)

Border styles.

Property	Type	Props
radius	string	

[color](#)

Color styles.

Property	Type	Props
background	string	
gradient	string	
text	string	

[spacing](#)

Spacing styles.

Property Type Props

margin object bottom, left, right, top

padding object bottom, left, right, top

typography

Typography styles.

Property Type Props

fontSize string

lineHeight string

First published

January 14, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Theme.json Version 1 Reference](#)”

[Previous Theme.json Version 2](#) [Previous: Theme.json Version 2](#)

[Next Migrating to Newer Versions](#) [Next: Migrating to Newer Versions](#)

Migrating to Newer Versions

In this article

Table of Contents

- [Migrating from v1 to v2](#)
 - [Renamed properties](#)
 - [New properties](#)
 - [Changes to property values](#)

[↑ Back to top](#)

This guide documents the changes between different theme.json versions and how to upgrade. Using older versions will continue to be supported. Upgrading is recommended because new development will continue in the newer versions.

Migrating from v1 to v2

Upgrading to v2 enables some new features and adjusts the naming of some old features to be more consistent with one another.

How to migrate from v1 to v2:

1. Update version to 2.
2. Rename the properties that were updated (see below) if you're using them.

Refer to the [dev note for the release](#) and the [reference documents](#) for the respective v1 and v2 versions.

[Renamed properties](#)

v1	v2
settings.border.customRadius	settings.border.radius
settings.spacing.customMargin	settings.spacing.margin
settings.spacing.customPadding	settings.spacing.padding
settings.typography.customLineHeight	settings.typography.lineHeight

[New properties](#)

New top-level properties: `customTemplates`, `templateParts`.

Additions to settings:

- `settings.appearanceTools`
- `settings.border.color`
- `settings.border.style`
- `settings.border.width`
- `settings.color.background`
- `settings.color.defaultGradients`
- `settings.color.defaultPalette`
- `settings.color.text`
- `settings.spacing.blockGap`
- `settings.typography.fontFamilies`
- `settings.typography.fontSize`
- `settings.typography.fontStyle`
- `settings.typography.fontWeight`
- `settings.typography.letterSpacing`
- `settings.typography.textColumns`
- `settings.typography.textDecoration`
- `settings.typography.textTransform`

Additions to styles:

- `styles.border.color`
- `styles.border.style`
- `styles.border.width`
- `styles.filter.duotone`
- `styles.spacing.blockGap`
- `styles.typography.fontSize`
- `styles.typography.fontStyle`
- `styles.typography.fontWeight`
- `styles.typography.letterSpacing`
- `styles.typography.textColumns`
- `styles.typography.textDecoration`
- `styles.typography.textTransform`

Changes to property values

The default font sizes provided by core (`settings.typography.fontSizes`) have been updated. The Normal and Huge sizes (with `normal` and `huge` slugs) have been removed from the list, and Extra Large (`x-large` slug) has been added. When the UI controls show the default values provided by core, Normal and Huge will no longer be present. However, their CSS classes and CSS Custom Properties are still enqueued to make sure existing content that uses them still works as expected.

First published

January 14, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Migrating to Newer Versions”](#)

[Previous Theme.json Version 1 Reference](#) [Previous: Theme.json Version 1 Reference](#)
[Next Available Styles Options](#) [Next: Available Styles Options](#)

Available Styles Options

[↑ Back to top](#)

New styles options are integrated into theme.json on a regular basis. Knowing the style options available through theme.json or the styles editor at any given time can be challenging. To clarify, the table below indicates the WordPress version when each theme.json styles option became available and when a corresponding control was added to the user interface to allow management of the style from the Styles editor.

Styles keys

Key	theme.json Since	Style Editor Since
<code>color.gradient</code>	5.8	5.9
<code>color.background</code>	5.8	5.9
<code>color.text</code>	5.8	5.9
<code>border.color</code>	5.9	5.9
<code>border.width</code>	5.9	5.9
<code>border.style</code>	5.9	5.9
<code>border.radius</code>	5.8	5.9
<code>border.radius.topLeft</code>	5.9	5.9
<code>border.radius.topRight</code>	5.9	5.9
<code>border.radius.bottomLeft</code>	5.9	5.9

Key	theme.json	Since	Style Editor	Since
border.radius.bottomRight	5.9		5.9	
border.top.color	6.1		6.1	
border.top.width	6.1		6.1	
border.top.style	6.1		6.1	
border.right.color	6.1		6.1	
border.right.width	6.1		6.1	
border.right.style	6.1		6.1	
border.bottom.color	6.1		6.1	
border.bottom.width	6.1		6.1	
border.bottom.style	6.1		6.1	
border.left.color	6.1		6.1	
border.left.width	6.1		6.1	
border.left.style	6.1		6.1	
typography.fontFamily	5.9		5.9	
typography.fontSize	5.8		5.9	
typography.fontStyle	5.9		5.9	
typography.fontWeight	5.9		5.9	
typography.letterSpacing	5.9		5.9	
typography.lineHeight	5.8		5.9	
typography.textDecoration	5.9		6.2	
typography.textTransform	5.9		6.0	
spacing.padding	5.9		5.9	
spacing.padding.top	5.8		5.9	
spacing.padding.right	5.8		5.9	
spacing.padding.left	5.8		5.9	
spacing.padding.bottom	5.8		5.9	
spacing.margin	5.9		5.9	
spacing.margin.top	5.8		5.9	
spacing.margin.right	5.8		5.9	
spacing.margin.left	5.8		5.9	
spacing.margin.bottom	5.8		5.9	
spacing.blockGap	5.9		5.9	
dimensions.minHeight	6.2		N/A	
outline.color	6.2		N/A	
outline.offset	6.2		N/A	
outline.style	6.2		N/A	
outline.width	6.2		N/A	
filter.duotone	5.9		N/A	
shadow	6.1		6.2	

First published

March 8, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Available Styles Options”](#)

[Previous Migrating to Newer Versions](#) [Previous: Migrating to Newer Versions](#)

[Next Component Reference](#) [Next: Component Reference](#)

Component Reference

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
 - [Popovers](#)
- [Docs & examples](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This package includes a library of generic WordPress components to be used for creating common UI elements shared between screens and features of the WordPress dashboard.

[Installation](#)

`npm install @wordpress/components --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.*

[Usage](#)

Within Gutenberg, these components can be accessed by importing from the `components` root directory:

```
/**  
 * WordPress dependencies  
 */  
import { Button } from '@wordpress/components';  
  
export default function MyButton() {  
    return <Button>Click Me!</Button>;  
}
```

Many components include CSS to add styles, which you will need to load in order for them to appear correctly. Within WordPress, add the `wp-components` stylesheet as a dependency of your plugin's stylesheet. See [wp_enqueue_style documentation](#) for how to specify dependencies.

In non-WordPress projects, link to the `build-style/style.css` file directly, it is located at `node_modules/@wordpress/components/build-style/style.css`.

[Popovers](#)

By default, the `Popover` component will render within an extra element appended to the body of the document.

If you want to precisely control where the popovers render, you will need to use the `Popover.Slot` component.

The following example illustrates how you can wrap a component using a `Popover` and have those popovers render to a single location in the DOM.

```
/**  
 * External dependencies  
 */  
import { Popover, SlotFillProvider } from '@wordpress/components';  
  
/**  
 * Internal dependencies  
 */  
import { MyComponentWithPopover } from './my-component';  
  
const Example = () => {  
    <SlotFillProvider>  
        <MyComponentWithPopover />  
        <Popover.Slot />  
    </SlotFillProvider>;  
};
```

[Docs & examples](#)

You can browse the components docs and examples at <https://wordpress.github.io/gutenberg/>

[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](#).

This package also has its own [contributing information](#) where you can find additional details.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Component Reference](#)

[Previous Available Styles Options](#) [Previous: Available Styles Options](#)

[Next AlignmentMatrixControl](#) [Next: AlignmentMatrixControl](#)

AlignmentMatrixControl

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [className](#)
 - [id](#)
 - [label](#)
 - [defaultValue](#)
 - [value](#)
 - [onChange](#)
 - [width](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

AlignmentMatrixControl components enable adjustments to horizontal and vertical alignments for UI.

[Usage](#)

```
import { useState } from 'react';
import { __experimentalAlignmentMatrixControl as AlignmentMatrixControl } from '@shopify/polaris';

const Example = () => {
  const [ alignment, setAlignment ] = useState( 'center center' );

  return (
    <AlignmentMatrixControl
      value={ alignment }
      onChange={ ( newAlignment ) => setAlignment( newAlignment ) } />
  );
}
```

```
) ;  
} ;
```

Props

The component accepts the following props:

className

The class that will be added to the classes of the underlying grid widget.

- Type: `string`
- Required: No

id

Unique ID for the component.

- Type: `string`
- Required: No

label

Accessible label. If provided, sets the `aria-label` attribute of the underlying grid widget.

- Type: `string`
- Required: No
- Default: Alignment Matrix Control

defaultValue

If provided, sets the default alignment value.

- Type: `AlignmentMatrixControlValue`
- Required: No
- Default: `center center`

value

The current alignment value.

- Type: `AlignmentMatrixControlValue`
- Required: No

onChange

A function that receives the updated alignment value.

- Type: `(newValue: AlignmentMatrixControlValue) => void`
- Required: No

width

If provided, sets the width of the control.

- Type: `number`
- Required: No
- Default: 92

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: AlignmentMatrixControl”](#)

[Previous Component Reference](#) [Previous: Component Reference](#)
[Next AnglePickerControl](#) [Next: AnglePickerControl](#)

AnglePickerControl

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - `label: string`
 - `value: number | string`
 - `onChange: (value: number) => void`
 - `_nextHasNoMarginBottom: boolean`

[↑ Back to top](#)

`AnglePickerControl` is a React component to render a UI that allows users to pick an angle.

Users can choose an angle in a visual UI with the mouse by dragging an angle indicator inside a circle or by directly inserting the desired angle in a text field.

Usage

```
import { useState } from 'react';
import { AnglePickerControl } from '@wordpress/components';

function Example() {
  const [ angle, setAngle ] = useState( 0 );
  return (
    <AnglePickerControl value={ angle } onChange={ setAngle } />
  );
}
```

```
<AnglePickerControl
    value={ angle }
    onChange={ setAngle }
    __nextHasNoMarginBottom
/>
);
};
```

Props

The component accepts the following props.

label: string

Label to use for the angle picker.

- Required: No
- Default: `__('Angle')`

value: number | string

The current value of the input. The value represents an angle in degrees and should be a value between 0 and 360.

- Required: Yes

onChange: (value: number) => void

A function that receives the new value of the input.

- Required: Yes

nextHasNoMarginBottom: boolean

Start opting into the new margin-free styles that will become the default in a future version, currently scheduled to be WordPress 6.4. (The prop can be safely removed once this happens.)

- Required: No
- Default: `false`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: AnglePickerControl](#)

[Previous AlignmentMatrixControl](#) [Previous: AlignmentMatrixControl](#)

Animate

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
- [Available Animation Types](#)
 - [appear](#)
 - [loading](#)
 - [slide-in](#)

[↑ Back to top](#)

Simple interface to introduce animations to components.

[Usage](#)

```
import { Animate, Notice } from '@wordpress/components';

const MyAnimatedNotice = () => (
    <Animate type="slide-in" options={{ origin: 'top' }}>
        { ( { className } ) => (
            <Notice className={ className } status="success">
                <p>Animation finished.</p>
            </Notice>
        ) }
    </Animate>
);


```

[Props](#)

Name	Type	Default	Description
type	string	undefined	Type of the animation to use.
options	object	{}	Options of the chosen animation.
children	function	undefined	A callback receiving a list of props (className) to apply to the DOM element to animate.

Available Animation Types

appear

This animation is meant for popover/modal content, such as menus appearing. It shows the height and width of the animated element scaling from 0 to full size, from its point of origin.

Options

Name	Type	Default	Description
origin	string	top center	Point of origin (top, bottom, middle right, left, center).

loading

This animation is meant to be used to indicate that activity is happening in the background. It is an infinitely-looping fade from 50% to full opacity. This animation has no options, and should be removed as soon as its relevant operation is completed.

slide-in

This animation is meant for sidebars and sliding menus. It shows the height and width of the animated element moving from a hidden position to its normal one.

Options

Name	Type	Default	Description
origin	string	left	Point of origin (left).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Animate”](#)

[Previous AnglePickerControl](#) [Previous: AnglePickerControl](#)

[Next Autocomplete](#) [Next: Autocomplete](#)

Autocomplete

In this article

[Table of Contents](#)

- [Props](#)
 - [record](#)
 - [onChange](#)
 - [onReplace](#)
 - [completers](#)
 - [contentRef](#)
 - [children](#)
 - [isSelected](#)
- [Autocompleters](#)
 - [The Completer Interface](#)
- [Usage](#)

[↑ Back to top](#)

This component is used to provide autocompletion support for a child input component.

[Props](#)

The following props are used to control the behavior of the component.

[record](#)

The rich text value object the autocomplete is being applied to.

- Required: Yes
- Type: `RichTextValue`

[onChange](#)

A function to be called when an option is selected to insert into the existing text.

- Required: Yes
- Type: `(value: string) => void`

[onReplace](#)

A function to be called when an option is selected to replace the existing text.

- Required: Yes
- Type: `(values: RichTextValue[]) => void`

completers

An array of all of the completers to apply to the current element.

- Required: Yes
- Type: `Array< WPCompleter >`

contentRef

A ref containing the editable element that will serve as the anchor for Autocomplete's Popover.

- Required: Yes
- `MutableRefObject< HTMLElement | undefined >`

children

A function that returns nodes to be rendered within the Autocomplete.

- Required: Yes
- Type: `Function`

isSelected

Whether or not the Autocomplete component is selected, and if its Popover should be displayed.

- Required: Yes
- Type: `Boolean`

Autocompleters

Autocompleters enable us to offer users options for completing text input. For example, Gutenberg includes a user autocomplete that provides a list of user names and completes a selection with a user mention like @mary.

Each completer declares:

- Its name.
- The text prefix that should trigger the display of completion options.
- Raw option data.
- How to render an option's label.
- An option's keywords, words that will be used to match an option with user input.
- What the completion of an option looks like, including whether it should be inserted in the text or used to replace the current block.

In addition, a completer may optionally declare:

- A class name to be applied to the completion menu.
- Whether it should apply to a specified text node.
- Whether the completer applies in a given context, defined via a Range before and a Range after the autocompletion trigger and query.

The Completer Interface

name

The name of the completer. Useful for identifying a specific completer to be overridden via extensibility hooks.

- Type: `String`
- Required: Yes

options

The raw options for completion. May be an array, a function that returns an array, or a function that returns a promise for an array.

Options may be of any type or shape. The completer declares how those options are rendered and what their completions should be when selected.

- Type: `Array | Function`
- Required: Yes

triggerPrefix

The string prefix that should trigger the completer. For example, Gutenberg's block completer is triggered when the '/' character is entered.

- Type: `String`
- Required: Yes

getOptionLabel

A function that returns the label for a given option. A label may be a string or a mixed array of strings, elements, and components.

- Type: `Function`
- Required: Yes

getOptionKeywords

A function that returns the keywords for the specified option.

- Type: `Function`
- Required: No

isOptionDisabled

A function that returns whether or not the specified option should be disabled. Disabled options cannot be selected.

- Type: `Function`
- Required: No

getOptionCompletion

A function that takes an option and responds with how the option should be completed. By default, the result is a value to be inserted in the text. However, a completer may explicitly declare how a completion should be treated by returning an object with `action` and `value` properties. The `action` declares what should be done with the `value`.

There are currently two supported actions:

- “insert-at-caret” – Insert the `value` into the text (the default completion action).
- “replace” – Replace the current block with the block specified in the `value` property.

allowContext

A function that takes a Range before and a Range after the autocomplete trigger and query text and returns a boolean indicating whether the completer should be considered for that context.

- Type: `Function`
- Required: No

className

A class name to apply to the autocomplete popup menu.

- Type: `String`
- Required: No

isDebounced

Whether to apply debouncing for the completer. Set to true to enable debouncing.

- Type: `Boolean`
- Required: No

Usage

The `Autocomplete` component is not currently intended to be used as a standalone component. It is used by other packages to provide autocomplete support for the block editor.

The block editor provides a separate, wrapped version of `Autocomplete` that supports the addition of custom completers via a filter.

To implement your own completer in the block editor you will:

1. Define the completer
2. Write a callback that will add your completer to the list of existing completers
3. Add a filter to the `editor.Autocomplete.completers` hook that will call your callback

The following example will add a contrived “fruits” completer to the block editor. Note that in the callback it’s possible to limit this new completer to a specific block type. In this case, our “fruits” completer will only be available from the “core/paragraph” block type.

```

( function () {
    // Define the completer
    const fruits = {
        name: 'fruit',
        // The prefix that triggers this completer
        triggerPrefix: '~',
        // The option data
        options: [
            { visual: '\ud83c\udc9e', name: 'Apple', id: 1 },
            { visual: '\ud83c\udc9f', name: 'Orange', id: 2 },
            { visual: '\ud83c\udc9d', name: 'Grapes', id: 3 },
            { visual: '\ud83c\udc9b', name: 'Mango', id: 4 },
            { visual: '\ud83c\udc9a', name: 'Strawberry', id: 5 },
            { visual: '\ud83c\udc9c', name: 'Blueberry', id: 6 },
            { visual: '\ud83c\udc99', name: 'Cherry', id: 7 },
        ],
        // Returns a label for an option like "\ud83c\udc9f Orange"
        getOptionLabel: ( option ) => `${ option.visual } ${ option.name }`,
        // Declares that options should be matched by their name
        getOptionKeywords: ( option ) => [ option.name ],
        // Declares that the Grapes option is disabled
        isOptionDisabled: ( option ) => option.name === 'Grapes',
        // Declares completions should be inserted as abbreviations
        getOptionCompletion: ( option ) => option.visual,
    };
}

// Define a callback that will add the custom completer to the list of
function appendTestCompleters( completers, blockName ) {
    return blockName === 'core/paragraph'
        ? [ ...completers, fruits ]
        : completers;
}

// Trigger our callback on the `editor.Autocomplete.completers` hook
wp.hooks.addFilter(
    'editor.Autocomplete.completers',
    'fruit-test/fruits',
    appendTestCompleters,
    11
);
} )();
}

```

Finally, enqueue your JavaScript file as you would any other, as in the following plugin example:

```

<?php
/**
 * Plugin Name: Fruit Autocompleter
 * Plugin URI: https://github.com/WordPress/gutenberg
 * Author: Gutenberg Team
 */

/**
 * Registers a custom script for the plugin.
 */

```

```
function enqueue_fruit_autocompleter_plugin_script() {
    wp_enqueue_script(
        'fruit-autocompleter',
        plugins_url( '/index.js', __FILE__ ),
        array(
            'wp-hooks',
        ),
    );
}

add_action( 'init', 'enqueue_fruit_autocompleter_plugin_script' );
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Autocomplete”](#)

[Previous Animate](#) [Previous: Animate](#)
[Next BaseControl](#) [Next: BaseControl](#)

BaseControl

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [id](#)
 - [label](#)
 - [hideLabelFromVision](#)
 - [help](#)
 - [className](#)
 - [children](#)
 - [nextHasNoMarginBottom](#)
- [BaseControl.VisualLabel](#)
- [Usage](#)
 - [Props](#)

[↑ Back to top](#)

BaseControl is a component used to generate labels and help text for components handling user inputs.

Usage

```
import { BaseControl, useBaseControlProps } from '@wordpress/components';

// Render a `BaseControl` for a textarea input
const MyCustomTextareaControl = ({ children, ...baseProps }) => (
    // `useBaseControlProps` is a convenience hook to get the props for the
    // and the inner control itself. Namely, it takes care of generating and
    // properly associating it with the `label` and `help` elements.
    const { baseControlProps, controlProps } = useBaseControlProps( baseProps )

    return (
        <BaseControl { ...baseControlProps } __nextHasNoMarginBottom={ true }>
            <textarea { ...controlProps }>
                { children }
            </textarea>
        </BaseControl>
    );
);
```

Props

The component accepts the following props:

id

The HTML `id` of the control element (passed in as a child to `BaseControl`) to which labels and help text are being generated. This is necessary to accessibly associate the label with that element.

The recommended way is to use the `useBaseControlProps` hook, which takes care of generating a unique `id` for you. Otherwise, if you choose to pass an explicit `id` to this prop, you are responsible for ensuring the uniqueness of the `id`.

- Type: `String`
- Required: No

label

If this property is added, a label will be generated using `label` property as the content.

- Type: `String`
- Required: No

hideLabelFromVision

If true, the label will only be visible to screen readers.

- Type: `Boolean`
- Required: No

help

Additional description for the control. It is preferable to use plain text for help, as it can be accessibly associated with the control using `aria-describedby`. When the help contains links, or otherwise non-plain text content, it will be associated with the control using `aria-details`.

- Type: `ReactNode`
- Required: No

className

Any other classes to add to the wrapper div.

- Type: `String`
- Required: No

children

The content to be displayed within the `BaseControl`.

- Type: `Element`
- Required: Yes

nextHasNoMarginBottom

Start opting into the new margin-free styles that will become the default in a future version.

- Type: `Boolean`
- Required: No
- Default: `false`

BaseControl.VisualLabel

`BaseControl.VisualLabel` is used to render a purely visual label inside a `BaseControl` component.

It should only be used in cases where the children being rendered inside `BaseControl` are already accessibly labeled, e.g., a button, but we want an additional visual label for that section equivalent to the labels `BaseControl` would otherwise use if the `label` prop was passed.

Usage

```
import { BaseControl } from '@wordpress/components';

const MyBaseControl = () => (
    <BaseControl help="This button is already accessibly labeled.">
        <BaseControl.VisualLabel>Author</BaseControl.VisualLabel>
        <Button>Select an author</Button>
    </BaseControl>
);
```

Props

className

Any other classes to add to the wrapper div.

- Type: `String`
- Required: No

children

The content to be displayed within the `BaseControl.VisualLabel`.

- Type: `Element`
- Required: Yes

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: BaseControl](#)

[Previous Autocomplete](#) [Previous: Autocomplete](#)
[Next BorderBoxControl](#) [Next: BorderBoxControl](#)

BorderBoxControl

In this article

Table of Contents

- [Development guidelines](#)
- [Usage](#)
- [Props](#)
 - [colors: \(PaletteObject | ColorObject \)\[\]](#)
 - [disableCustomColors: boolean](#)
 - [enableAlpha: boolean](#)
 - [enableStyle: boolean](#)
 - [hideLabelFromVision: boolean](#)
 - [label: string](#)
 - [onChange: \(value?: Object \) => void](#)
 - [popoverPlacement: string](#)
 - [popoverOffset: number](#)
 - [size: string](#)
 - [value: Object](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

This component provides users with the ability to configure a single “flat” border or separate borders per side.

Development guidelines

The `BorderBoxControl` effectively has two view states. The first, a “linked” view, allows configuration of a flat border via a single `BorderControl`.

The second, a “split” view, contains a `BorderControl` for each side as well as a visualizer for the currently selected borders. Each view also contains a button to toggle between the two.

When switching from the “split” view to “linked”, if the individual side borders are not consistent, the “linked” view will display any border properties selections that are consistent while showing a mixed state for those that aren’t. For example, if all borders had the same color and style but different widths, then the border dropdown in the “linked” view’s `BorderControl` would show that consistent color and style but the “linked” view’s width input would show “Mixed” placeholder text.

Usage

```
import { useState } from 'react';
import { __experimentalBorderBoxControl as BorderBoxControl } from '@wordpress/block-editor';
import { __ } from '@wordpress/i18n';

const colors = [
  { name: 'Blue 20', color: '#72aee6' },
  // ...
];

const MyBorderBoxControl = () => {
  const defaultBorder = {
    color: '#72aee6',
    style: 'dashed',
    width: '1px',
  };
  const [ borders, setBorders ] = useState( {
    top: defaultBorder,
    right: defaultBorder,
    bottom: defaultBorder,
    left: defaultBorder,
  } );
  const onChange = ( newBorders ) => setBorders( newBorders );

  return (
    <BorderBoxControl
```

```
        colors={ colors }
        label={ __( 'Borders' ) }
        onChange={ onChange }
        value={ borders }
    />
);
};
```

If you're using this component outside the editor, you can [ensure Tooltip positioning](#) for the BorderBoxControl's color and style options, by rendering your BorderBoxControl with a Popover.Slot further up the element tree and within a SlotFillProvider overall.

Props

colors: (PaletteObject | ColorObject)[]

An array of color definitions. This may also be a multi-dimensional array where colors are organized by multiple origins.

Each color may be an object containing a name and color value.

- Required: No
- Default: []

disableCustomColors: boolean

This toggles the ability to choose custom colors.

- Required: No

enableAlpha: boolean

This controls whether the alpha channel will be offered when selecting custom colors.

- Required: No
- Default: false

enableStyle: boolean

This controls whether to support border style selections.

- Required: No
- Default: true

hideLabelFromVision: boolean

Provides control over whether the label will only be visible to screen readers.

- Required: No

label: string

If provided, a label will be generated using this as the content.

Whether it is visible only to screen readers is controlled via hideLabelFromVision.

- Required: No

onChange: (value?: Object) => void

A callback function invoked when any border value is changed. The value received may be a “flat” border object, one that has properties defining individual side borders, or `undefined`.

Note: This will be undefined if a user clears all borders.

- Required: Yes

popoverPlacement: string

The position of the color popovers relative to the control wrapper.

By default, popovers are displayed relative to the button that initiated the popover. By supplying a popover placement, you force the popover to display in a specific location.

The available base placements are ‘top’, ‘right’, ‘bottom’, ‘left’. Each of these base placements has an alignment in the form `-start` and `-end`. For example, ‘right-start’, or ‘bottom-end’. These allow you to align the tooltip to the edges of the button, rather than centering it.

- Required: No

popoverOffset: number

The space between the popover and the control wrapper.

- Required: No

size: string

Size of the control.

- Required: No
- Default: `default`
- Allowed values: `default`, `__unstable-large`

value: Object

An object representing the current border configuration.

This may be a “flat” border where the object has `color`, `style`, and `width` properties or a “split” border which defines the previous properties but for each side; `top`, `right`, `bottom`, and `left`.

Examples:

```
const flatBorder = { color: '#72aee6', style: 'solid', width: '1px' };
const splitBorders = {
  top: { color: '#72aee6', style: 'solid', width: '1px' },
  right: { color: '#e65054', style: 'dashed', width: '2px' },
  bottom: { color: '#68de7c', style: 'solid', width: '1px' },
  left: { color: '#f2d675', style: 'dotted', width: '1em' },
};
```

- Required: No

First published

March 24, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: BorderBoxControl](#)

[Previous BaseControl](#) [Previous: BaseControl](#)

[Next BorderControl](#) [Next: BorderControl](#)

BorderControl

In this article

[Table of Contents](#)

- [Development guidelines](#)
- [Usage](#)
- [Props](#)
 - [colors: \(PaletteObject | ColorObject \)\[\]](#)
 - [disableCustomColors: boolean](#)
 - [disableUnits: boolean](#)
 - [enableAlpha: boolean](#)
 - [enableStyle: boolean](#)
 - [hideLabelFromVision: boolean](#)
 - [isCompact: boolean](#)
 - [label: string](#)
 - [onChange: \(value?: Object \) => void](#)
 - [shouldSanitizeBorder: boolean](#)
 - [showDropdownHeader: boolean](#)
 - [size: string](#)
 - [value: Object](#)
 - [width: CSSProperties\['width' \]](#)
 - [withSlider: boolean](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

This component provides control over a border’s color, style, and width.

Development guidelines

The `BorderControl` brings together internal sub-components which allow users to set the various properties of a border. The first sub-component, a `BorderDropdown` contains options representing border color and style. The border width is controlled via a `UnitControl` and an optional `RangeControl`.

Border radius is not covered by this control as it may be desired separate to color, style, and width. For example, the border radius may be absorbed under a “shape” abstraction.

Usage

```
import { useState } from 'react';
import { __experimentalBorderControl as BorderControl } from '@wordpress/core-data';
import { __ } from '@wordpress/i18n';

const colors = [
  { name: 'Blue 20', color: '#72aee6' },
  // ...
];

const MyBorderControl = () => {
  const [ border, setBorder ] = useState();

  return (
    <BorderControl
      colors={ colors }
      label={ __( 'Border' ) }
      onChange={ setBorder }
      value={ border }
    />
  );
};


```

If you’re using this component outside the editor, you can [ensure Tooltip positioning](#)

for the `BorderControl`’s color and style options, by rendering your `BorderControl` with a `Popover.Slot` further up the element tree and within a `SlotFillProvider` overall.

Props

`colors: (PaletteObject | ColorObject)[]`

An array of color definitions. This may also be a multi-dimensional array where colors are organized by multiple origins.

Each color may be an object containing a `name` and `color` value.

- Required: No
- Default: []

disableCustomColors: boolean

This toggles the ability to choose custom colors.

- Required: No

disableUnits: boolean

This controls whether unit selection should be disabled.

- Required: No

enableAlpha: boolean

This controls whether the alpha channel will be offered when selecting custom colors.

- Required: No
- Default: `false`

enableStyle: boolean

This controls whether to support border style selection.

- Required: No
- Default: `true`

hideLabelFromVision: boolean

Provides control over whether the label will only be visible to screen readers.

- Required: No

isCompact: boolean

This flags the `BorderControl` to render with a more compact appearance. It restricts the width of the control and prevents it from expanding to take up additional space.

- Required: No

label: string

If provided, a label will be generated using this as the content.

Whether it is visible only to screen readers is controlled via hideLabelFromVision.

- Required: No

onChange: (value?: Object)=> void

A callback function invoked when the border value is changed via an interaction that selects or clears, border color, style, or width.

Note: the value may be undefined if a user clears all border properties.

- Required: Yes

shouldSanitizeBorder: boolean

If opted into, sanitizing the border means that if no width or color have been selected, the border style is also cleared and undefined is returned as the new border value.

- Required: No
- Default: true

showDropdownHeader: boolean

Whether or not to render a header for the border color and style picker dropdown. The header includes a label for the color picker and a close button.

- Required: No

size: string

Size of the control.

- Required: No
- Default: default
- Allowed values: default, __unstable-large

value: Object

An object representing a border or undefined. Used to set the current border configuration for this component.

Example:

```
{  
  color: '#72aee6',  
  style: 'solid',  
  width: '2px',  
}
```

- Required: No

[width: CSSProperties\['width' \]](#)

Controls the visual width of the BorderControl. It has no effect if the `isCompact` prop is set to `true`.

- Required: No

[withSlider: boolean](#)

Flags whether this BorderControl should also render a RangeControl for additional control over a border's width.

- Required: No

First published

March 24, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: BorderControl](#)

[Previous BorderBoxControl](#) [Previous: BorderBoxControl](#)

[Next BoxControl](#) [Next: BoxControl](#)

BoxControl

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [allowReset: boolean](#)
 - [splitOnAxis: boolean](#)
 - [inputProps: object](#)
 - [label: string](#)
 - [onChange: \(next: BoxControlValue\) => void](#)
 - [resetValues: object](#)
 - [sides: string\[\]](#)
 - [units: WPUnitControlUnit\[\]](#)
 - [values: object](#)
 - [onMouseOver: function](#)
 - [onMouseOut: function](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

BoxControl components let users set values for Top, Right, Bottom, and Left. This can be used as an input control for values like padding or margin.

Usage

```
import { useState } from 'react';
import { __experimentalBoxControl as BoxControl } from '@wordpress/components';

const Example = () => {
    const [ values, setValues ] = useState( {
        top: '50px',
        left: '10%',
        right: '10%',
        bottom: '50px',
    } );
    return (
        <BoxControl
            values={ values }
            onChange={ ( nextValues ) => setValues( nextValues ) }
        />
    );
};

export default Example;
```

Props

allowReset: boolean

If this property is true, a button to reset the box control is rendered.

- Required: No
- Default: `true`

splitOnAxis: boolean

If this property is true, when the box control is unlinked, vertical and horizontal controls can be used instead of updating individual sides.

- Required: No
- Default: `false`

inputProps: object

Props for the internal [UnitControl](#) components.

- Required: No
- Default: `{ min: 0 }`

label: string

Heading label for the control.

- Required: No
- Default: __('Box Control')

onChange: (next: BoxControlValue) => void

A callback function when an input value changes.

- Required: Yes

resetValues: object

The top, right, bottom, and left box dimension values to use when the control is reset.

- Required: No
- Default: { top: undefined, right: undefined, bottom: undefined, left: undefined }

sides: string[]

Collection of sides to allow control of. If omitted or empty, all sides will be available. Allowed values are “top”, “right”, “bottom”, “left”, “vertical”, and “horizontal”.

- Required: No

units: WPUnitControlUnit[]

Collection of available units which are compatible with [UnitControl](#).

- Required: No

values: object

The top, right, bottom, and left box dimension values.

- Required: No

onMouseOver: function

A handler for onMouseOver events.

- Required: No

onMouseOut: function

A handler for onMouseOut events.

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: BoxControl](#)

[Previous BorderControl](#) [Previous: BorderControl](#)

[Next BaseField](#) [Next: BaseField](#)

BaseField

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [disabled: boolean](#)
 - [hasError: boolean](#)
 - [isInline: boolean](#)
 - [isSubtle: boolean](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

BaseField is an internal (i.e., not exported in the `index.js`) primitive component used for building more complex fields like `TextField`. It provides error handling and focus styles for field components. It does *not* handle layout of the component aside from wrapping the field in a `Flex` wrapper.

[Usage](#)

BaseField is primarily used as a hook rather than a component:

```
function useExampleField( props ) {  
  const { as = 'input', ...baseProps } = useBaseField( props );  
  
  const inputProps = {  
    as,  
    // more cool stuff here  
  };  
  
  return { inputProps, ...baseProps };  
}
```

```
function ExampleField( props, forwardRef ) {
  const { preFix, affix, disabled, inputProps, ...baseProps } =
    useExampleField( props );

  return (
    <View { ...baseProps } disabled={ disabled }>
      { preFix }
      <View autocomplete="off" { ...inputProps } disabled={ disabled }>
        { affix }
      </View>
    );
}
```

[Props](#)

[disabled: boolean](#)

Whether the field is disabled.

- Required: No

[hasError: boolean](#)

Renders an error style around the component.

- Required: No
- Default: `false`

[isInline: boolean](#)

Renders a component that can be inlined in some text.

- Required: No
- Default: `false`

[isSubtle: boolean](#)

Renders a subtle variant of the component.

- Required: No
- Default: `false`

First published

May 28, 2021

Last updated

February 9, 2023

Edit article

[Improve it on GitHub: BaseField”](#)

ButtonGroup

In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
 - [Best practices](#)
 - [States](#)
- [Development guidelines](#)
 - [Usage](#)
- [Related components](#)

[↑ Back to top](#)

ButtonGroup can be used to group any related buttons together. To emphasize related buttons, a group should share a common container.

25% 50% 75% 100%

Design guidelines

Usage

Selected action



25% 50% 75% 100%

Do

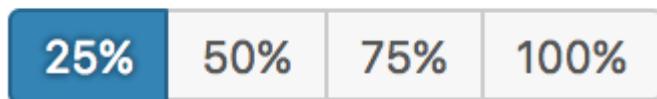
Only one option in a button group can be selected and active at a time. Selecting one option deselects any other.

Best practices

Button groups should:

- **Be clearly and accurately labeled.**
- **Clearly communicate that clicking or tapping will trigger an action.**
- **Use established colors appropriately.** For example, only use red buttons for actions that are difficult or impossible to undo.
- **Have consistent locations in the interface.**

States



Active and available button groups

A button group's state makes it clear which button is active. Hover and focus states express the available selection options for buttons in a button group.

Disabled button groups

Button groups that cannot be selected can either be given a disabled state, or be hidden.

Development guidelines

Usage

```
import { Button, ButtonGroup } from '@wordpress/components';

const MyButtonGroup = () => (
    <ButtonGroup>
        <Button variant="primary">Button 1</Button>
        <Button variant="primary">Button 2</Button>
    </ButtonGroup>
);
```

Related components

- For individual buttons, use a `Button` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ButtonGroup](#)

[Previous BaseField](#) [Previous: BaseField](#)

[Next Button](#) [Next: Button](#)

Button

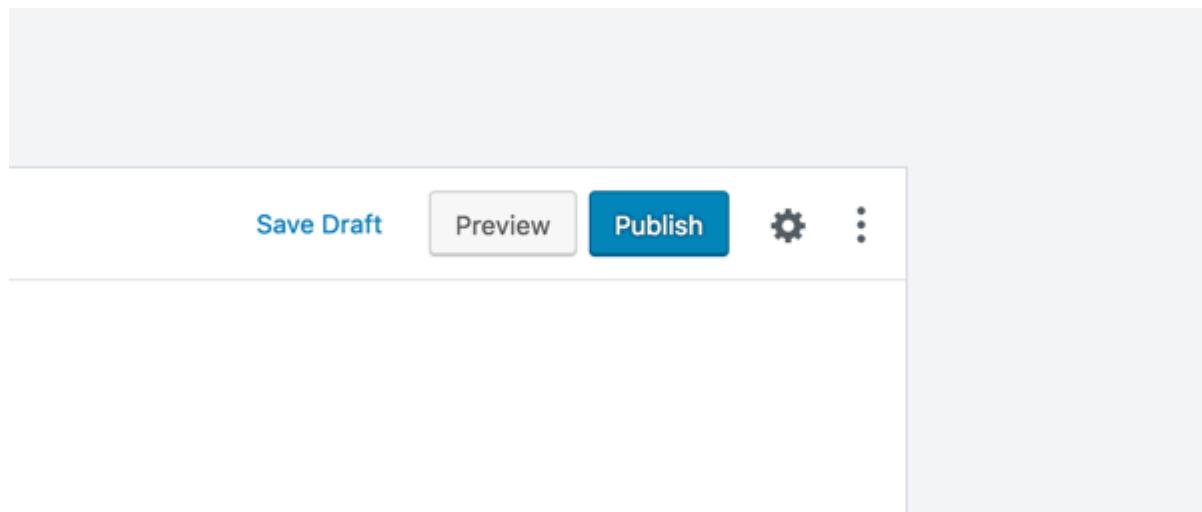
In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
 - [Best practices](#)
 - [Content guidelines](#)
 - [Types](#)
 - [Hierarchy](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

Buttons let users take actions and make choices with a single click or tap.



Design guidelines

Usage

Buttons tell users what actions they can take and give them a way to interact with the interface. You'll find them throughout a UI, particularly in places like:

- Modals
- Forms
- Toolbars

Best practices

Buttons should:

- **Be clearly and accurately labeled.**
- **Clearly communicate that clicking or tapping will trigger an action.**
- **Use established colors appropriately.** For example, only use red buttons for actions that are difficult or impossible to undo.
- **Prioritize the most important actions.** This helps users focus. Too many calls to action on one screen can be confusing, making users unsure what to do next.
- **Have consistent locations in the interface.**

Content guidelines

Buttons should be clear and predictable—users should be able to anticipate what will happen when they click a button. Never deceive a user by mislabeling a button.

Buttons text should lead with a strong verb that encourages action, and add a noun that clarifies what will actually change. The only exceptions are common actions like Save, Close, Cancel, or OK. Otherwise, use the {verb}+{noun} format to ensure that your button gives the user enough information.

Button text should also be quickly scannable — avoid unnecessary words and articles like the, an, or a.

Types

Link button

Link buttons have low emphasis. They don't stand out much on the page, so they're used for less-important actions. What's less important can vary based on context, but it's usually a supplementary action to the main action we want someone to take. Link buttons are also useful when you don't want to distract from the content.

[Add new category](#)

Default button

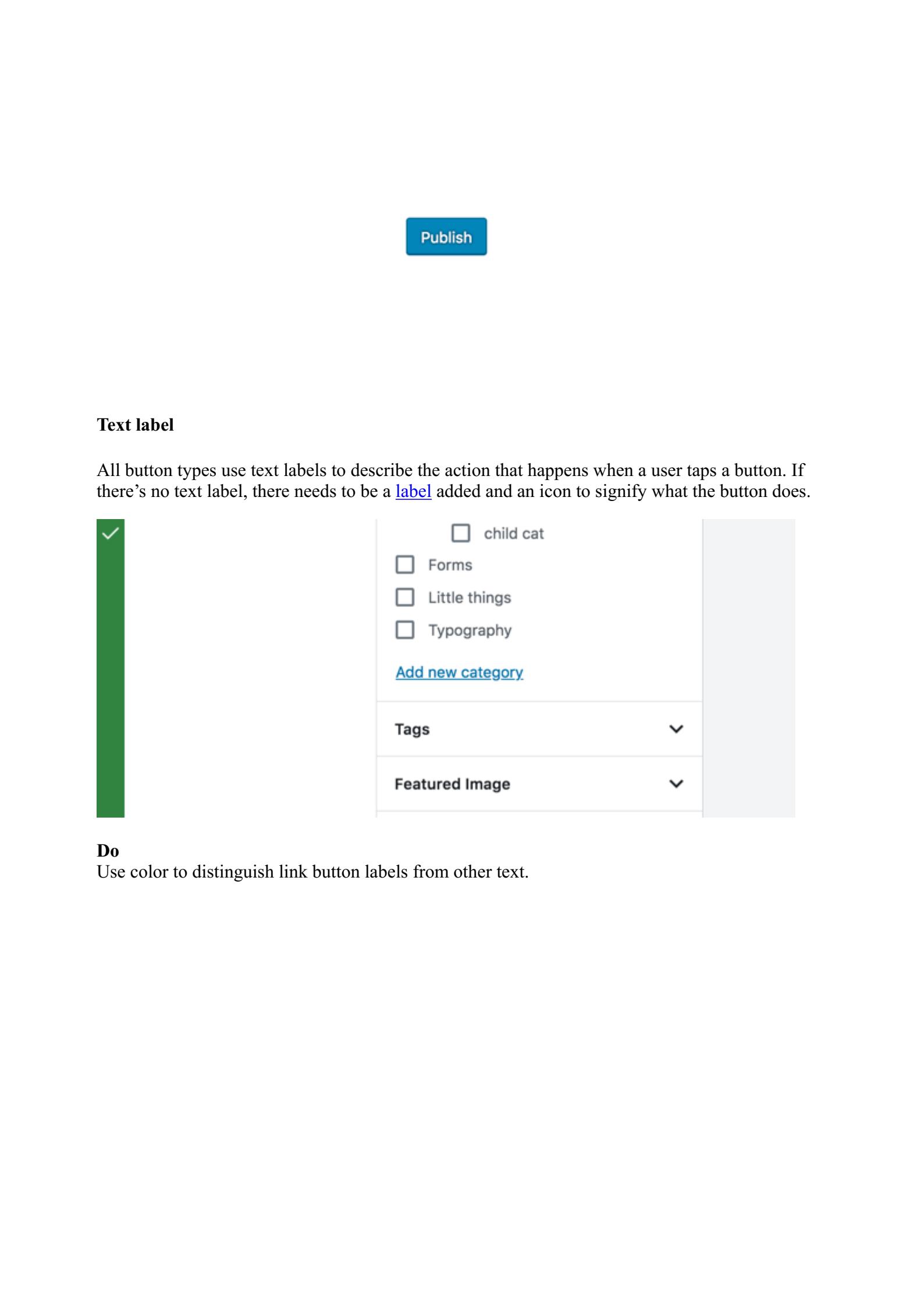
Default buttons have medium emphasis. The button appearance helps differentiate them from the page background, so they're useful when you want more emphasis than a link button offers.

Preview

Primary button

Primary buttons have high emphasis. Their color fill and shadow means they pop off the background.

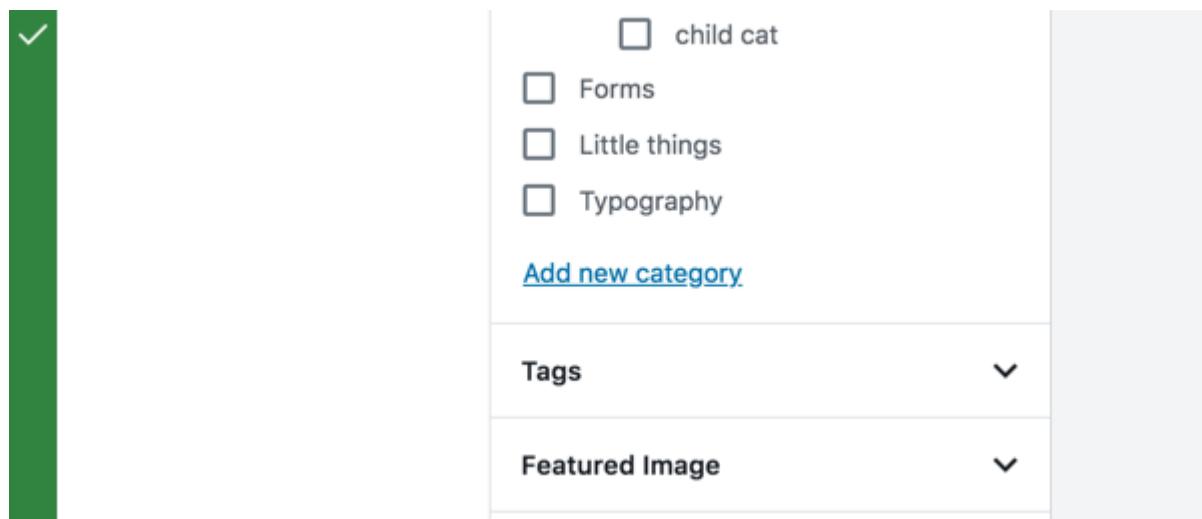
Since a high-emphasis button commands the most attention, a layout should contain a single primary button. This makes it clear that other buttons have less importance and helps users understand when an action requires their attention.



Publish

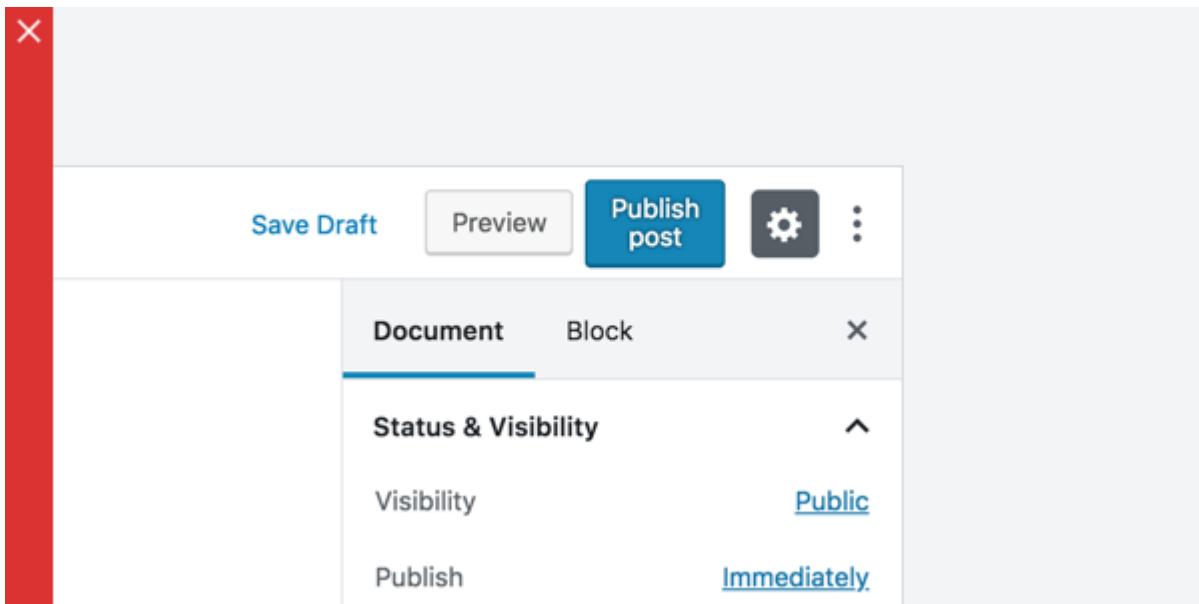
Text label

All button types use text labels to describe the action that happens when a user taps a button. If there's no text label, there needs to be a [label](#) added and an icon to signify what the button does.



Do

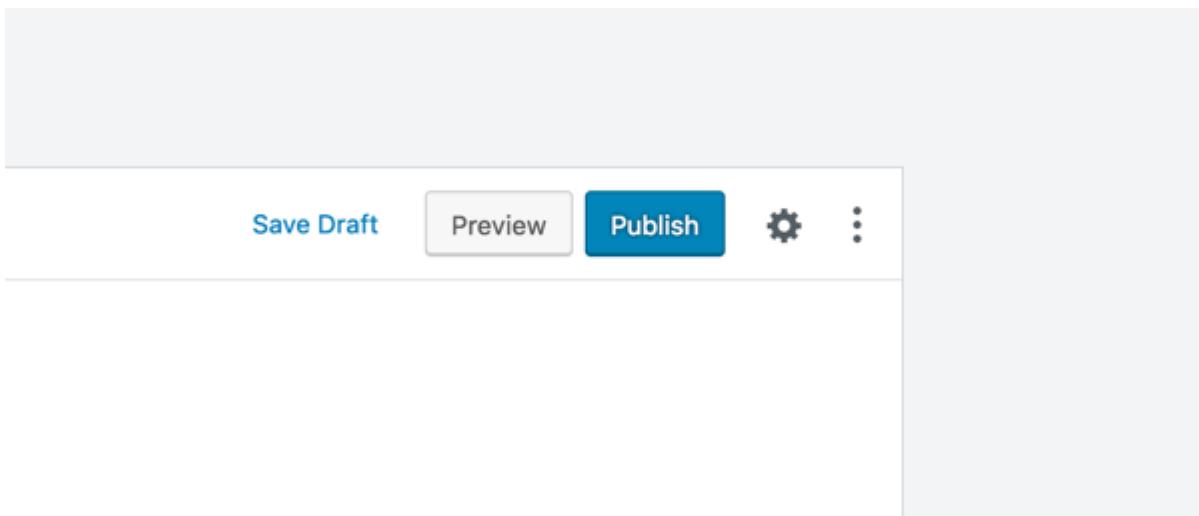
Use color to distinguish link button labels from other text.



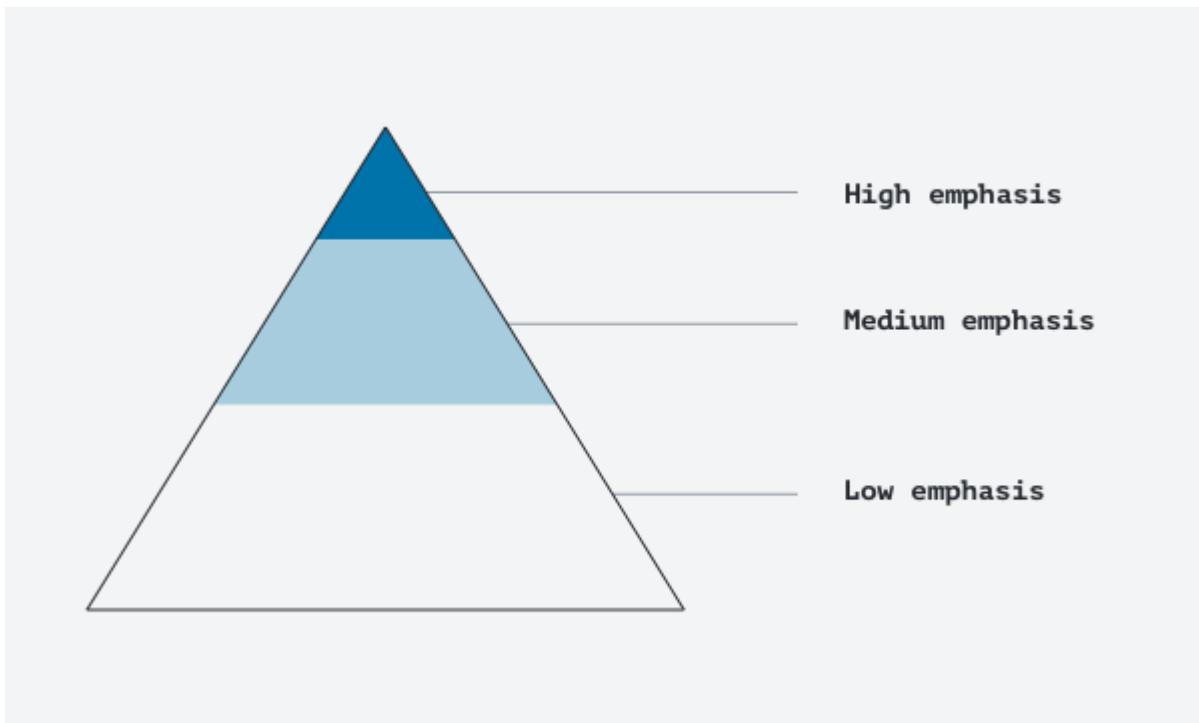
Don't

Don't wrap button text. For maximum legibility, keep text labels on a single line.

Hierarchy



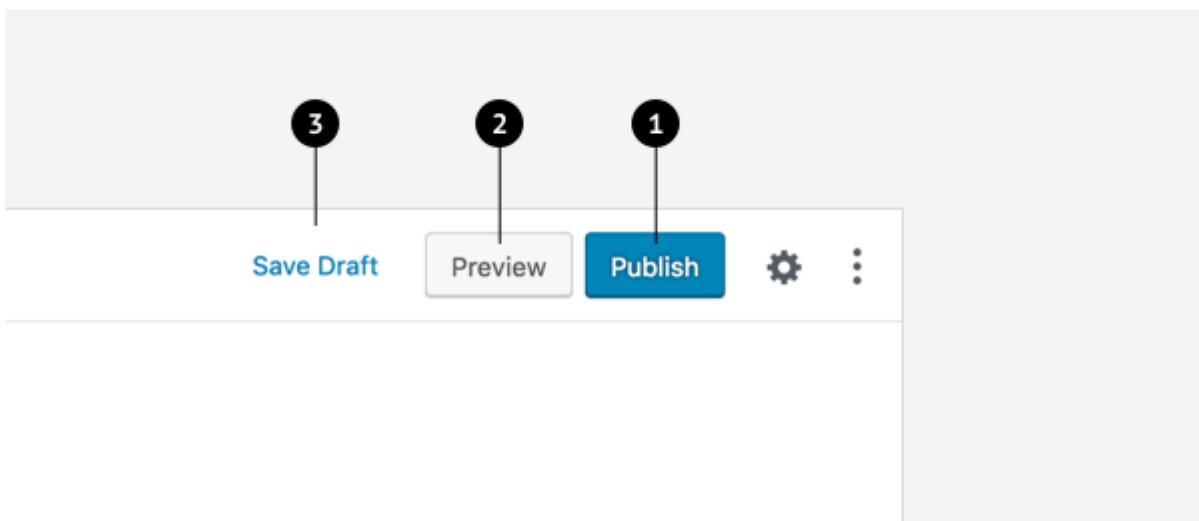
A layout should contain a single prominently-located button. If multiple buttons are required, a single high-emphasis button can be joined by medium- and low-emphasis buttons mapped to less-important actions. When using multiple buttons, make sure the available state of one button doesn't look like the disabled state of another.



A button's level of emphasis helps determine its appearance, typography, and placement.

Placement

Use button types to express different emphasis levels for all the actions a user can perform.



This screen layout uses:

1. A primary button for high emphasis.
2. A default button for medium emphasis.
3. A link button for low emphasis.

Placement best practices:

- **Do:** When using multiple buttons in a row, show users which action is more important by placing it next to a button with a lower emphasis (e.g. a primary button next to a default button, or a default button next to a link button).

- **Don't:** Don't place two primary buttons next to one another — they compete for focus. Only use one primary button per view.
- **Don't:** Don't place a button below another button if there is space to place them side by side.
- **Caution:** Avoid using too many buttons on a single page. When designing pages in the app or website, think about the most important actions for users to take. Too many calls to action can cause confusion and make users unsure what to do next — we always want users to feel confident and capable.

Development guidelines

Usage

Renders a button with default style.

```
import { Button } from '@wordpress/components';

const MyButton = () => <Button variant="secondary">Click me!</Button>;
```

Props

The presence of a `href` prop determines whether an `anchor` element is rendered instead of a `button`.

Props not included in this set will be applied to the `a` or `button` element.

`children: ReactNode`

The button's children.

- Required: No

`className: string`

An optional additional class name to apply to the rendered button.

- Required: No

`describedBy: string`

An accessible description for the button.

- Required: No

`disabled: boolean`

Whether the button is disabled. If `true`, this will force a `button` element to be rendered, even when an `href` is given.

- Required: No

`href: string`

If provided, renders a instead of button.

- Required: No

`icon: IconProps< unknown >['icon']`

If provided, renders an [Icon](#) component inside the button.

- Required: No

`iconPosition: 'left' | 'right'`

If provided with icon, sets the position of icon relative to the text. Available options are left|right.

- Required: No
- Default: left

`iconSize: IconProps< unknown >['size']`

If provided with icon, sets the icon size. Please refer to the [Icon](#) component for more details regarding the default value of its size prop.

- Required: No

`isBusy: boolean`

Indicates activity while a action is being performed.

- Required: No

`isDestructive: boolean`

Renders a red text-based button style to indicate destructive behavior.

- Required: No

`isLink: boolean`

Deprecated: Renders a button with an anchor style.

Use variant prop with link value instead.

- Required: No
- Default: false

`isPressed: boolean`

Renders a pressed button style.

If the native `aria-pressed` attribute is also set, it will take precedence.

- Required: No

`isPrimary: boolean`

Deprecated: Renders a primary button style.

Use `variant` prop with `primary` value instead.

- Required: No
- Default: `false`

`isSecondary: boolean`

Deprecated: Renders a default button style.

Use `variant` prop with `secondary` value instead.

- Required: No
- Default: `false`

`isSmall: boolean`

Decreases the size of the button.

Deprecated in favor of the `size` prop. If both props are defined, the `size` prop will take precedence.

- Required: No

`isTertiary: boolean`

Deprecated: Renders a text-based button style.

Use `variant` prop with `tertiary` value instead.

- Required: No
- Default: `false`

`label: string`

Sets the `aria-label` of the component, if none is provided. Sets the Tooltip content if `showTooltip` is provided.

- Required: No

`shortcut: string | { display: string; ariaLabel: string; }`

If provided with `showTooltip`, appends the Shortcut label to the tooltip content. If an object is provided, it should contain `display` and `ariaLabel` keys.

- Required: No

`showTooltip: boolean`

If provided, renders a [Tooltip](#) component for the button.

- Required: No

`size: 'default' | 'compact' | 'small'`

The size of the button.

- `'default'`: For normal text-label buttons, unless it is a toggle button.
- `'compact'`: For toggle buttons, icon buttons, and buttons when used in context of either.
- `'small'`: For icon buttons associated with more advanced or auxiliary features.

If the deprecated `isSmall` prop is also defined, this prop will take precedence.

- Required: No
- Default: `'default'`

`target: string`

If provided with `href`, sets the `target` attribute to the `a`.

- Required: No

`text: string`

If provided, displays the given text inside the button. If the button contains children elements, the text is displayed before them.

- Required: No

`tooltipPosition: PopoverProps['position']`

If provided with `showTooltip`, sets the position of the tooltip. Please refer to the [Tooltip](#) component for more details regarding the defaults.

- Required: No

`variant: 'primary' | 'secondary' | 'tertiary' | 'link'`

Specifies the button's style. The accepted values are `'primary'` (the primary button styles), `'secondary'` (the default button styles), `'tertiary'` (the text-based button styles), and `'link'` (the link button styles).

- Required: No

Related components

- To group buttons together, use the [ButtonGroup](#) component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Button”](#)

[Previous ButtonGroup](#) [Previous: ButtonGroup](#)

[Next CardBody](#) [Next: CardBody](#)

CardBody

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [isScrollable: boolean](#)
 - [isShady: boolean](#)
 - [size: string](#)

[↑ Back to top](#)

CardBody renders an optional content area for a [Card](#). Multiple CardBody components can be used within Card if needed.

[Usage](#)

```
import { Card, CardBody } from '@wordpress/components';

const Example = () => (
    <Card>
        <CardBody>...</CardBody>
    </Card>
);
```

[Props](#)

Note: This component is connected to [Card's Context](#). The value of the size prop is derived from the Card parent component (if there is one). Setting this prop directly on this component will override any derived values.

[isScrollable: boolean](#)

Determines if the component is scrollable.

- Required: No
- Default: `false`

[isShady: boolean](#)

Renders with a light gray background color.

- Required: No
- Default: `false`

[size: string](#)

Determines the amount of padding within the component.

- Required: No
- Default: `medium`
- Allowed values: `xSmall`, `small`, `medium`, `large`

First published

June 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CardBody](#)”

[Previous Button](#) [Previous: Button](#)
[Next CardDivider](#) [Next: CardDivider](#)

CardDivider

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [Inherited props](#)

[↑ Back to top](#)

CardDivider renders an optional divider within a [Card](#). It is typically used to divide multiple [CardBody](#) components from each other.

Usage

```
import { Card, CardBody, CardDivider } from '@wordpress/components';

const Example = () => (
  <Card>
    <CardBody>...</CardBody>
    <CardDivider />
    <CardBody>...</CardBody>
  </Card>
);
```

Props

Inherited props

CardDivider inherits all of the [Divider props](#).

First published

June 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CardDivider](#)

[Previous CardBody](#) [Previous: CardBody](#)

[Next CardFooter](#) [Next: CardFooter](#)

CardFooter

In this article

Table of Contents

- [Usage](#)
 - [Flex](#)
- [Props](#)
 - [isBorderless: boolean](#)
 - [isShady: boolean](#)
 - [justify: CSSProperties\[‘justifyContent’ \]](#)
 - [size: string](#)

[↑ Back to top](#)

CardFooter renders an optional footer within a [Card](#).

Usage

```
import { Card, CardFooter } from '@wordpress/components';

const Example = () => (
  <Card>
    <CardBody>...</CardBody>
    <CardFooter>...</CardFooter>
  </Card>
);
```

Flex

Underneath, `CardFooter` uses the [Flex layout component](#). This improves the alignment of child items within the component.

```
import {
  Button,
  Card,
  CardFooter,
  FlexItem,
  FlexBlock,
} from '@wordpress/components';

const Example = () => (
  <Card>
    <CardBody>...</CardBody>
    <CardFooter>
      <FlexBlock>Content</FlexBlock>
      <FlexItem>
        <Button>Action</Button>
      </FlexItem>
    </CardFooter>
  </Card>
);
```

Check out [the documentation](#) on Flex for more details on layout composition.

Props

Note: This component is connected to [Card's Context](#). The value of the `size` and `isBorderless` props is derived from the `Card` parent component (if there is one). Setting these props directly on this component will override any derived values.

isBorderless: boolean

Renders without a border.

- Required: No
- Default: `false`

[isShady: boolean](#)

Renders with a light gray background color.

- Required: No
- Default: `false`

[justify: CSSProperties\[‘justifyContent’ \]](#)

See the documentation for the `justify` prop for the [Flex component](#)

[size: string](#)

Determines the amount of padding within the component.

- Required: No
- Default: `medium`
- Allowed values: `xSmall`, `small`, `medium`, `large`

First published

June 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CardFooter”](#)

[Previous CardDivide](#) [Previous: CardDivide](#)
[Next CardHeader](#) [Next: CardHeader](#)

CardHeader

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [isBorderless: boolean](#)
 - [isShady: boolean](#)
 - [size: string](#)

[↑ Back to top](#)

CardHeader renders an optional header within a [Card](#).

Usage

```
import { Card, CardHeader } from '@wordpress/components';

const Example = () => (
  <Card>
    <CardHeader>...</CardHeader>
    <CardBody>...</CardBody>
  </Card>
);
```

Props

Note: This component is connected to [Card's Context](#). The value of the `size` and `isBorderless` props is derived from the `Card` parent component (if there is one). Setting these props directly on this component will override any derived values.

isBorderless: boolean

Renders without a border.

- Required: No
- Default: `false`

isShady: boolean

Renders with a light gray background color.

- Required: No
- Default: `false`

size: string

Determines the amount of padding within the component.

- Required: No
- Default: `medium`
- Allowed values: `xSmall`, `small`, `medium`, `large`

First published

June 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CardHeader](#)

CardMedia

In this article

Table of Contents

- [Usage](#)
- [Placement](#)

[↑ Back to top](#)

CardMedia provides a container for full-bleed content within a [Card](#), such as images, video, or even just a background color.

[Usage](#)

```
import { Card, CardBody, CardMedia } from '@wordpress/components';

const Example = () => (
  <Card>
    <CardMedia>
      
    </CardMedia>
    <CardBody>...</CardBody>
  </Card>
);
```

[Placement](#)

CardMedia can be placed in any order as a direct child of a Card (it can also exist as the only child component). The styles will automatically round the corners of the inner media element.

First published

June 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CardMedia”](#)

[Previous CardHeader](#) [Previous: CardHeader](#)
[Next Card](#) [Next: Card](#)

Card

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [elevation: number](#)
 - [isBorderless: boolean](#)
 - [isRounded: boolean](#)
 - [size: string](#)
 - [Inherited props](#)
- [Sub-Components](#)
 - [Sub-Components Example](#)
 - [Context](#)

[↑ Back to top](#)

Card provides a flexible and extensible content container.

[Usage](#)

Card also provides a convenient set of [sub-components](#) such as CardBody, CardHeader, CardFooter, and more (see below).

```
import {  
    Card,  
    CardHeader,  
    CardBody,  
    CardFooter,  
    __experimentalText as Text,  
    __experimentalHeading as Heading,  
} from '@wordpress/components';  
  
function Example() {  
    return (  
        <Card>  
            <CardHeader>  
                <Heading level={ 4 }>Card Title</Heading>  
            </CardHeader>  
            <CardBody>  
                <Text>Card Content</Text>  
            </CardBody>  
            <CardFooter>  
                <Text>Card Footer</Text>  
            </CardFooter>  
        </Card>  
    );  
}
```

Props

elevation: number

Size of the elevation shadow, based on the Style system's elevation system. This may be helpful in highlighting certain content. For more information, check out [Elevation](#).

- Required: No
- Default: 0

isBorderless: boolean

Renders without a border.

- Required: No
- Default: false

isRounded: boolean

Renders with rounded corners.

- Required: No
- Default: true

size: string

Determines the amount of padding within the component.

- Required: No
- Default: medium
- Allowed values: xSmall, small, medium, large

Inherited props

Card also inherits all of the [Surface props](#).

Sub-Components

This component provides a collection of sub-component that can be used to compose various interfaces.

- [`<CardBody />`](#)
- [`<CardDivider />`](#)
- [`<CardFooter />`](#)
- [`<CardHeader />`](#)
- [`<CardMedia />`](#)

Sub-Components Example

```
import {
  Card,
  CardBody,
```

```
CardDivider,  
CardFooter,  
CardHeader,  
CardMedia,  
} from '@wordpress/components';  
  
const Example = () => (  
  <Card>  
    <CardHeader>...</CardHeader>  
    <CardBody>...</CardBody>  
    <CardDivider />  
    <CardBody>...</CardBody>  
    <CardMedia>  
        
    </CardMedia>  
    <CardFooter>...</CardFooter>  
  </Card>  
) ;
```

Context

<Card />'s sub-components are connected to <Card /> using [Context](#). Certain props like `size` and `isBorderless` are passed through to some of the sub-components.

In the following example, the <CardBody /> will render with a size of `small`:

```
import { Card,CardBody } from '@wordpress/components';  
  
const Example = () => (  
  <Card size="small">  
    <CardBody>...</CardBody>  
  </Card>  
) ;
```

These sub-components are designed to be flexible. The Context props can be overridden by the sub-component(s) as required. In the following example, the last <CardBody /> will render it's specified size:

```
import { Card,CardBody } from '@wordpress/components';  
  
const Example = () => (  
  <Card size="small">  
    <CardBody>...</CardBody>  
    <CardBody>...</CardBody>  
    <CardBody size="large">...</CardBody>  
  </Card>  
) ;
```

First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: Card”](#)

[Previous CardMedia](#) [Previous: CardMedia](#)
[Next CheckboxControl](#) [Next: CheckboxControl](#)

CheckboxControl

In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

Checkboxes allow the user to select one or more items from a set.



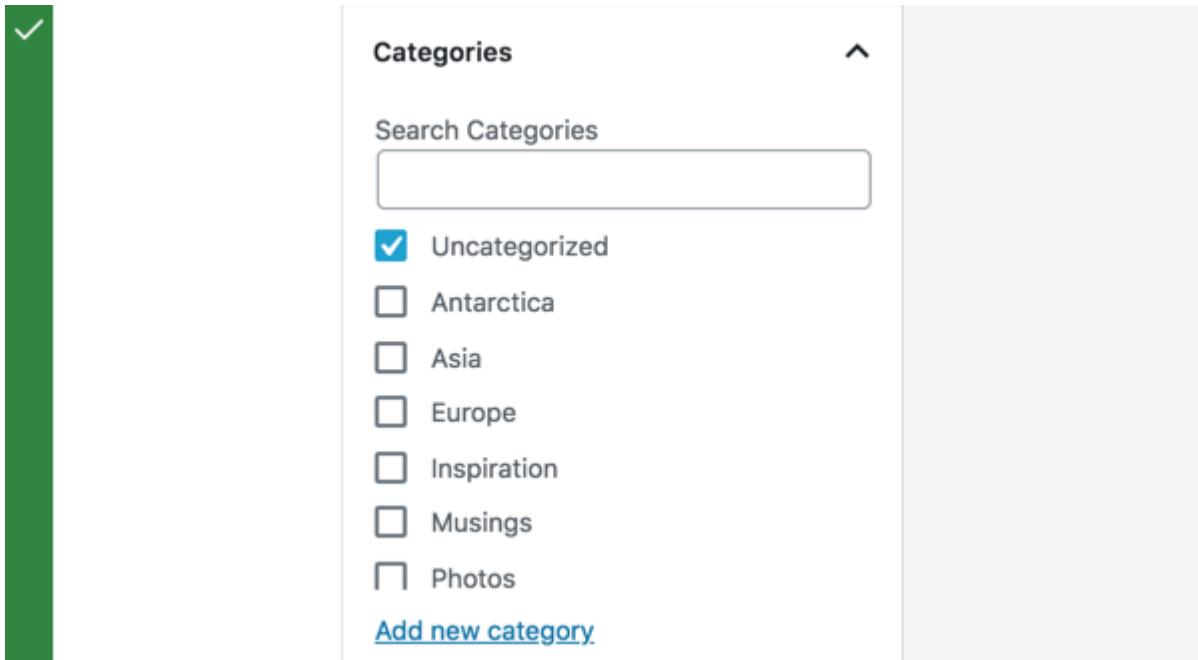
Design guidelines

Usage

When to use checkboxes

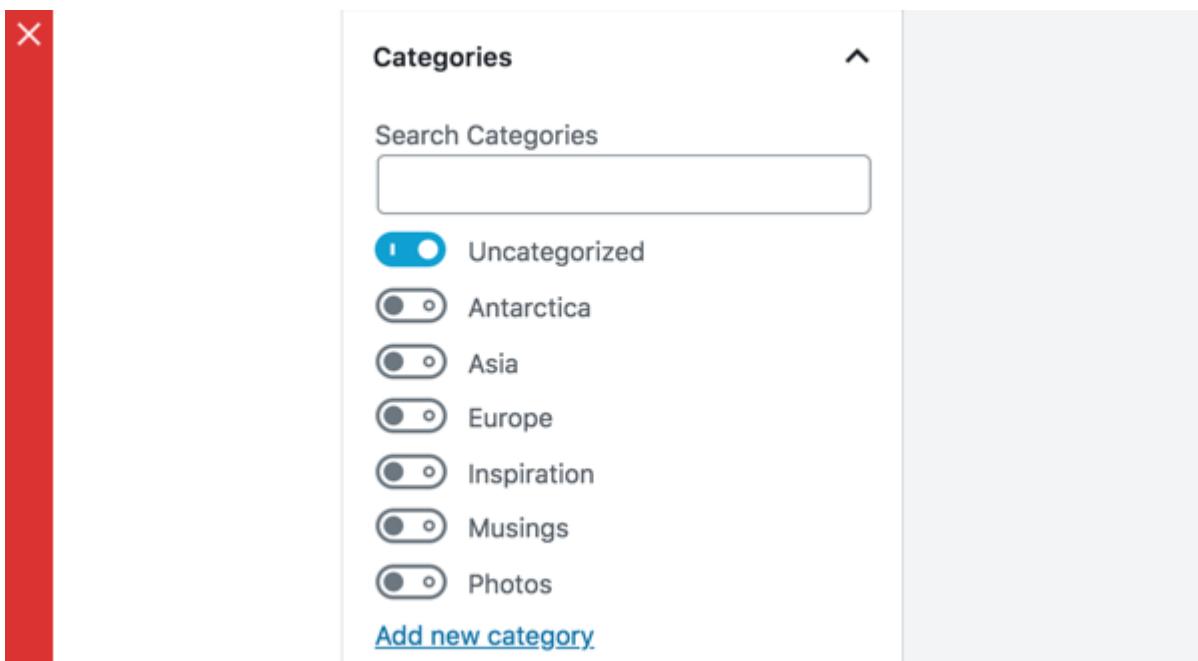
Use checkboxes when you want users to:

- Select one or multiple items from a list.
- Open a list containing sub-selections.



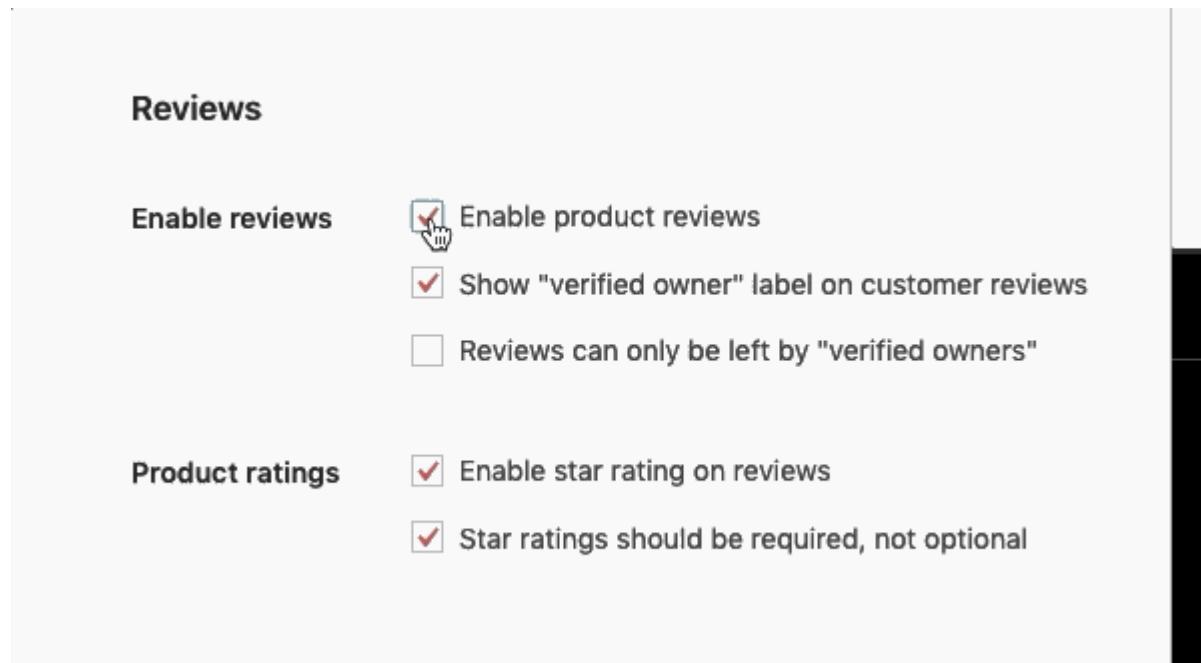
Do

Use checkboxes when users can select multiple items from a list. They let users select more than one item.



Don't

Don't use toggles when a list consists of multiple options. Use checkboxes — they take up less space.



Checkboxes can be used to open a list containing sub-selections.

Parent and child checkboxes

Checkboxes can have a parent-child relationship, with secondary options nested under primary options.

Title	Author	Categories	Tags
<input checked="" type="checkbox"/> Blogging with Simplenote	Melissa	Simplenote, Tips	—
<input checked="" type="checkbox"/> Bring All Your Notes Home with the New Importer	Dan	Linux, macOS, Releases, Simplenote, Windows	—
<input checked="" type="checkbox"/> Desktop App Updates: Find Your Focus	Dan	Android, macOS, Releases, Simplenote, Windows	—

When the parent checkbox is *checked*, all the child checkboxes are checked. When a parent checkbox is *unchecked*, all the child checkboxes are unchecked.

Title	Author	Categories	Tags
<input type="checkbox"/> Blogging with Simplenote	Melissa	Simplenote , Tips	—
<input type="checkbox"/> Bring All Your Notes Home with the New Importer	Dan	Linux , macOS , Releases , Simplenote , Windows	—
<input checked="" type="checkbox"/> Desktop App Updates: Find Your Focus	Dan	Android , macOS , Releases , Simplenote , Windows	—

If only a few child checkboxes are checked, the parent checkbox becomes a mixed checkbox.

Development guidelines

Usage

Render an is author checkbox:

```
import { useState } from 'react';
import { CheckboxControl } from '@wordpress/components';

const MyCheckboxControl = () => {
    const [ isChecked, setChecked ] = useState( true );
    return (
        <CheckboxControl
            label="Is author"
            help="Is the user a author or not?"
            checked={ isChecked }
            onChange={ setChecked }
        />
    );
};
```

Props

The set of props accepted by the component will be specified below.
Props not included in this set will be applied to the input element.

label: string|false

A label for the input field, that appears at the side of the checkbox.
The prop will be rendered as content a label element.

If no prop is passed an empty label is rendered.
If the prop is set to false no label is rendered.

- Required: No

`help: string|Element`

If this property is added, a help text will be generated using help property as the content.

- Required: No

`checked: boolean`

If checked is true the checkbox will be checked. If checked is false the checkbox will be unchecked.

If no value is passed the checkbox will be unchecked.

- Required: No

`onChange: function`

A function that receives the checked state (boolean) as input.

- Required: Yes

`indeterminate: boolean`

If indeterminate is true the state of the checkbox will be indeterminate.

- Required: No

[Related components](#)

- To select one option from a set, and you want to show all the available options at once, use the `RadioControl` component.
- To toggle a single setting on or off, use the `FormToggle` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CheckboxControl](#)

[Previous Card](#) [Previous: Card](#)

[Next CircularOptionPicker](#) [Next: CircularOptionPicker](#)

CircularOptionPicker

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [className: string](#)
 - [actions: ReactNode](#)
 - [options: ReactNode](#)
 - [children: ReactNode](#)
 - [asButtons: boolean](#)
 - [loop: boolean](#)
- [Subcomponents](#)
 - [CircularOptionPicker.ButtonAction](#)
 - [CircularOptionPicker DropdownLinkAction](#)

[↑ Back to top](#)

This component is not exported, and therefore can only be used internally to the `@wordpress/components` package.

`CircularOptionPicker` is a component that displays a set of options as circular buttons.

[Usage](#)

```
import { useState } from 'react';
import { CircularOptionPicker } from '../circular-option-picker';

const Example = () => {
  const [ currentColor, setCurrentColor ] = useState();
  const colors = [
    { color: '#f00', name: 'Red' },
    { color: '#0f0', name: 'Green' },
    { color: '#00f', name: 'Blue' },
  ];
  const colorOptions = (
    <>
      { colors.map( ( { color, name }, index ) => {
        return (
          <CircularOptionPicker.Option
            key={`${color}-${index}`}
            tooltipText={name}
            style={ { backgroundColor: color, color } }
            isSelected={ index === currentColor }
            onClick={ () => setCurrentColor( index ) }
            aria-label={name}
          />
        );
      }) }
  );
}
```

```

        } ) )
    </>
);
return (
    <CircularOptionPicker
        options={ colorOptions }
        actions={
            <CircularOptionPicker.ButtonAction
                onClick={ () => setCurrentColor( undefined ) }
            >
                { 'Clear' }
            </CircularOptionPicker.ButtonAction>
        }
    />
);
}
;

```

Props

className: string

A CSS class to apply to the wrapper element.

- Required: No

actions: ReactNode

The action(s) to be rendered after the options, such as a ‘clear’ button as seen in `ColorPalette`.

Usually a `CircularOptionPicker.ButtonAction` or `CircularOptionPicker DropdownLinkAction` component.

- Required: No

options: ReactNode

The options to be rendered, such as color swatches.

Usually a `CircularOptionPicker.Option` component.

- Required: No

children: ReactNode

The child elements.

- Required: No

[asButtons: boolean](#)

Whether the control should present as a set of buttons, each with its own tab stop.

- Required: No
- Default: `false`

[loop: boolean](#)

Prevents keyboard interaction from wrapping around. Only used when `asButtons` is not true.

- Required: No
- Default: `true`

[Subcomponents](#)

[CircularOptionPicker.ButtonAction](#)

A `ButtonAction` is an action that is rendered as a button alongside the options themselves.

A common use case is a ‘clear’ button to deselect the currently selected option.

Props

`className: string`

A CSS class to apply to the underlying `Button` component.

- Required: No

`children: ReactNode`

The button’s children.

- Required: No

Inherited props

`CircularOptionPicker.ButtonAction` also inherits all of the [Button props](#), except for `href` and `target`.

[CircularOptionPicker.DropdownLinkAction](#)

`CircularOptionPicker.DropdownLinkAction` is an action that’s hidden behind a dropdown toggle. The button is formatted as a link and rendered as an `anchor` element.

Props

`className: string`

A CSS class to apply to the underlying `Dropdown` component.

- Required: No

`linkText: string`

The text to be displayed on the button.

- Required: Yes

`dropdownProps: object`

The props for the underlying `Dropdown` component.

Inherits all of the [Dropdown props](#), except for `className` and `renderToggle`.

- Required: Yes

`buttonProps: object`

Props for the underlying `Button` component.

Inherits all of the [Button props](#), except for `href`, `target`, and `children`.

- Required: No

First published

February 23, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CircularOptionPicker”](#)

[Previous CheckboxControl](#) [Previous: CheckboxControl](#)
[Next ClipboardButton](#) [Next: ClipboardButton](#)

ClipboardButton

In this article

Table of Contents

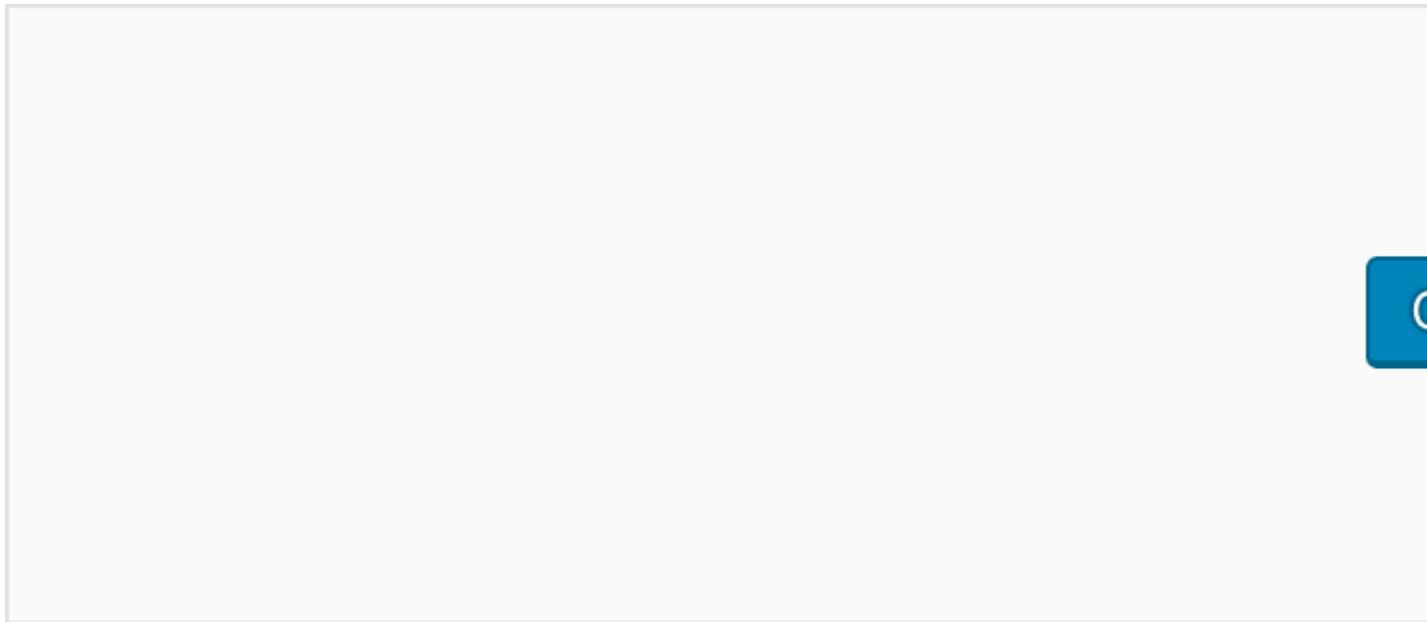
- [Usage](#)

- [Props](#)
 - [className](#)
 - [text](#)
 - [onCopy](#)
 - [onFinishCopy](#)
 - [Inherited props](#)

[↑ Back to top](#)

This component is deprecated. Please use the `useCopyToClipboard` hook from the `@wordpress/compose` package instead.

With a clipboard button, users copy text (or other elements) with a single click or tap.



[Usage](#)

```
import { useState } from 'react';
import { ClipboardButton } from '@wordpress/components';

const MyClipboardButton = () => {
    const [ hasCopied, setHasCopied ] = useState( false );
    return (
        <ClipboardButton
            variant="primary"
            text="Text to be copied."
            onCopy={ () => setHasCopied( true ) }
            onFinishCopy={ () => setHasCopied( false ) }
        >
            { hasCopied ? 'Copied!' : 'Copy Text' }
        </ClipboardButton>
    );
};
```

Props

The component accepts the following props:

className

The class that will be added to the classes of the underlying <Button> component.

- Type: `string`
- Required: no

text

The text that will be copied to the clipboard.

- Type: `string`
- Required: yes

onCopy

The function that will be called when the text is copied.

- Type: `() => void`
- Required: yes

onFinishCopy

The function that will be called when the text is copied and the copy animation is finished.

- Type: `() => void`
- Required: no

Inherited props

Any additional props will be passed the underlying <Button/> component. See the [Button](#) component for more details on the available props.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ClipboardButton](#)

[Previous CircularOptionPicker](#) [Previous: CircularOptionPicker](#)
[Next ColorIndicator](#) [Next: ColorIndicator](#)

ColorIndicator

In this article

Table of Contents

- [Single component](#)
- [Used in sidebar](#)
- [Usage](#)
- [Props](#)
 - [className: string](#)
 - [colorValue: CSSProperties\['background' \]](#)

[↑ Back to top](#)

ColorIndicator is a React component that renders a specific color in a circle. It's often used to summarize a collection of used colors in a child component.

[Single component](#)



[Used in sidebar](#)

Messages

[Usage](#)

```
import { ColorIndicator } from '@wordpress/components';
const MyColorIndicator = () => <ColorIndicator colorValue="#0073aa" />;
```

[Props](#)

The component accepts the following props:

[className: string](#)

Extra classes for the used `` element. By default only `component-color-indicator` is added.

- Required: No

[colorValue: CSSProperties\['background' \]](#)

The color of the indicator. Any value from the CSS [background](#) property is supported.

- Required: Yes

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ColorIndicator](#)”

[Previous ClipboardButton](#) [Previous: ClipboardButton](#)

[Next ColorPalette](#) [Next: ColorPalette](#)

ColorPalette

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [clearable: boolean](#)
 - [colors: PaletteObject\[\] | ColorObject\[\]](#)
 - [disableCustomColors: boolean](#)
 - [enableAlpha: boolean](#)
 - [headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | '1' | '2' | '3' | '4' | '5' | '6'](#)
 - [value: string](#)
 - [onChange: OnColorChange](#)
 - [asButtons: boolean](#)
 - [loop: boolean](#)

[↑ Back to top](#)

ColorPalette allows the user to pick a color from a list of pre-defined color entries.

[Usage](#)

```
import { useState } from 'react';
import { ColorPalette } from '@wordpress/components';

const MyColorPalette = () => {
  const [ color, setColor ] = useState ( '#f00' )
  const colors = [
```

```

        { name: 'red', color: '#f00' },
        { name: 'white', color: '#fff' },
        { name: 'blue', color: '#00f' },
    ];

    return (
        <ColorPalette
            colors={ colors }
            value={ color }
            onChange={ ( color ) => setColor( color ) }
        />
    );
} );

```

If you're using this component outside the editor, you can

[ensure Tooltip positioning](#)

for the ColorPalette's color swatches, by rendering your ColorPalette with a Popover.Slot further up the element tree and within a SlotFillProvider overall.

Props

The component accepts the following props.

clearable: boolean

Whether the palette should have a clearing button.

- Required: No
- Default: true

colors: PaletteObject[] | ColorObject[]

Array with the colors to be shown. When displaying multiple color palettes to choose from, the format of the array changes from an array of colors objects, to an array of color palettes.

- Required: No
- Default: []

disableCustomColors: boolean

Whether to allow the user to pick a custom color on top of the predefined choices (defined via the colors prop).

- Required: No
- Default: false

enableAlpha: boolean

This controls whether the alpha channel will be offered when selecting custom colors.

- Required: No

- Default: `false`

[headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’](#)

The heading level.

- Required: No
- Default: 2

[value: string](#)

Currently active value.

- Required: No

[onChange: OnColorChange](#)

Callback called when a color is selected.

- Required: Yes

[asButtons: boolean](#)

Whether the control should present as a set of buttons, each with its own tab stop.

- Required: No
- Default: `false`

[loop: boolean](#)

Prevents keyboard interaction from wrapping around. Only used when `asButtons` is not true.

- Required: No
- Default: `true`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ColorPalette”](#)

[Previous ColorIndicator](#) [Previous: ColorIndicator](#)
[Next ColorPicker](#) [Next: ColorPicker](#)

ColorPicker

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [color: string](#)
 - [onChange: \(hex8Color: string\) => void](#)
 - [enableAlpha: boolean](#)
 - [defaultValue: string | undefined](#)
 - [copyFormat: 'hex' | 'hsl' | 'rgb' | undefined](#)

[↑ Back to top](#)

ColorPicker is a color picking component based on `react-colorful`. It lets you pick a color visually or by manipulating the individual RGB(A), HSL(A) and Hex(8) color values.

[Usage](#)

```
import { useState } from 'react';
import { ColorPicker } from '@wordpress/components';

function Example() {
    const [color, setColor] = useState();
    return (
        <ColorPicker
            color={color}
            onChange={setColor}
            enableAlpha
            defaultValue="#000"
        />
    );
}
```

[Props](#)

[color: string](#)

The current color value to display in the picker. Must be a hex or hex8 string.

- Required: No

[onChange: \(hex8Color: string\) => void](#)

Fired when the color changes. Always passes a hex or hex8 color string.

- Required: No

[enableAlpha: boolean](#)

When `true` the color picker will display the alpha channel both in the bottom inputs as well as in the color picker itself.

- Required: No
- Default: `false`

[defaultValue: string | undefined](#)

An optional default value to use for the color picker.

- Required: No
- Default: `'#fff'`

[copyFormat: 'hex' | 'hsl' | 'rgb' | undefined](#)

The format to copy when clicking the displayed color format.

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ColorPicker](#)

[Previous ColorPalette](#) [Previous: ColorPalette](#)

[Next ComboboxControl](#) [Next: ComboboxControl](#)

ComboboxControl

In this article

Table of Contents

- [Design guidelines](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

ComboboxControl is an enhanced version of a [SelectControl](#), with the addition of being able to search for options using a search input.

Design guidelines

These are the same as [the ones for SelectControls](#), but this component is better suited for when there are too many items to scroll through or load at once so you need to filter them based on user input.

Development guidelines

Usage

```
import { useState } from 'react';
import { ComboboxControl } from '@wordpress/components';

const options = [
    {
        value: 'small',
        label: 'Small',
    },
    {
        value: 'normal',
        label: 'Normal',
    },
    {
        value: 'large',
        label: 'Large',
    },
];
function MyComboboxControl() {
    const [ fontSize, setFontSize ] = useState();
    const [ filteredOptions, setFilteredOptions ] = useState( options );
    return (
        <ComboboxControl
            label="Font Size"
            value={ fontSize }
            onChange={ setFontSize }
            options={ filteredOptions }
            onFilterValueChange={ ( inputValue ) =>
                setFilteredOptions(
                    options.filter( ( option ) =>
                        option.value === inputValue
                    )
                )
            }
        />
    );
}
```

Props

label

The label for the control.

- Type: `String`
- Required: Yes

hideLabelFromVision

If true, the label will only be visible to screen readers.

- Type: `Boolean`
- Required: No

help

If this property is added, a help text will be generated using help property as the content.

- Type: `String`
- Required: No

options

The options that can be chosen from.

- Type: `Array<{ value: string, label: string }>`
- Required: Yes

onFilterValueChange

Function called when the control's search input value changes. The argument contains the next input value.

- Type: `(value: string) => void`
- Required: No

onChange

Function called with the selected value changes.

- Type: `(value: string | null | undefined) => void`
- Required: No

value

The current value of the control.

- Type: `string | null`
- Required: No

experimentalRenderItem

Custom renderer invoked for each option in the suggestion list. The render prop receives as its argument an object containing, under the `item` key, the single option's data (directly from the array of data passed to the `options` prop).

- Type: `(args: { item: object }) => ReactNode`
- Required: No

Related components

- Like this component, but without a search input, the `CustomSelectControl` component.
- To select one option from a set, when you want to show all the available options at once, use the `Radio` component.
- To select one or more items from a set, use the `CheckboxControl` component.
- To toggle a single setting on or off, use the `ToggleControl` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ComboboxControl](#)

[Previous ColorPicker](#) [Previous: ColorPicker](#)
[Next ConfirmDialog](#) [Next: ConfirmDialog](#)

ConfirmDialog

In this article

Table of Contents

- [Usage](#)
 - [Uncontrolled mode](#)
 - [Controlled mode](#)
 - [Unsupported: Multiple instances](#)
- [Custom Types](#)
- [Props](#)
 - [title: string](#)
 - [children: React.ReactNode](#)
 - [isOpen: boolean](#)
 - [onConfirm: \(event: DialogInputEvent \) => void](#)
 - [onCancel: \(event: DialogInputEvent \) => void](#)

- [confirmButtonText: string](#)
- [cancelButtonText: string](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`ConfirmDialog` is built of top of [Modal](#) and displays a confirmation dialog, with *confirm* and *cancel* buttons.

The dialog is confirmed by clicking the *confirm* button or by pressing the `Enter` key. It is cancelled (closed) by clicking the *cancel* button, by pressing the `ESC` key, or by clicking outside the dialog focus (i.e, the overlay).

Usage

`ConfirmDialog` has two main implicit modes: controlled and uncontrolled.

Uncontrolled mode

Allows the component to be used standalone, just by declaring it as part of another React’s component render method:

- It will be automatically open (displayed) upon mounting;
- It will be automatically closed when clicking the *cancel* button, by pressing the `ESC` key, or by clicking outside the dialog focus (i.e, the overlay);
- `onCancel` is not mandatory but can be passed. Even if passed, the dialog will still be able to close itself.

Activating this mode is as simple as omitting the `isOpen` prop. The only mandatory prop, in this case, is the `onConfirm` callback. The message is passed as the `children`. You can pass any JSX you’d like, which allows to further format the message or include sub-component if you’d like:

```
import { __experimentalConfirmDialog as ConfirmDialog } from '@wordpress/c'

function Example() {
  return (
    <ConfirmDialog onConfirm={ () => console.debug( ' Confirmed! ' ) }>
      Are you sure? <strong>This action cannot be undone!</strong>
    </ConfirmDialog>
  );
}
```

Controlled mode

Let the parent component control when the dialog is open/closed. It’s activated when a boolean value is passed to `isOpen`:

- It will not be automatically closed. You need to let it know when to open/close by updating the value of the `isOpen` prop;

- Both `onConfirm` and the `onCancel` callbacks are mandatory props in this mode;
- You'll want to update the state that controls `isOpen` by updating it from the `onCancel` and `onConfirm` callbacks.

```
import { useState } from 'react';
import { __experimentalConfirmDialog as ConfirmDialog } from '@wordpress/c

function Example() {
    const [ isOpen, setIsOpen ] = useState( true );

    const handleConfirm = () => {
        console.debug( 'Confirmed!' );
        setIsOpen( false );
    };

    const handleCancel = () => {
        console.debug( 'Cancelled!' );
        setIsOpen( false );
    };

    return (
        <ConfirmDialog
            isOpen={ isOpen }
            onConfirm={ handleConfirm }
            onCancel={ handleCancel }
        >
            Are you sure? <strong>This action cannot be undone!</strong>
        </ConfirmDialog>
    );
}
```

Unsupported: Multiple instances

Multiple `ConfirmDialogs` is an edge case that's currently not officially supported by this component. At the moment, new instances will end up closing the last instance due to the way the `Modal` is implemented.

Custom Types

```
type DialogInputEvent =
    | KeyboardEvent< HTMLDivElement >
    | MouseEvent< HTMLButtonElement >;
```

Props

`title: string`

- Required: No

An optional `title` for the dialog. Setting a title will render it in a title bar at the top of the dialog, making it a bit taller. The bar will also include an `x` close button at the top-right corner.

[children: React.ReactNode](#)

- Required: Yes

The actual message for the dialog. It's passed as children and any valid `ReactNode` is accepted:

```
<ConfirmDialog>
  Are you sure? <strong>This action cannot be undone!</strong>
</ConfirmDialog>
```

[isOpen: boolean](#)

- Required: No

Defines if the dialog is open (displayed) or closed (not rendered/displayed). It also implicitly toggles the controlled mode if set or the uncontrolled mode if it's not set.

[onConfirm: \(event: DialogInputEvent \) => void](#)

- Required: Yes

The callback that's called when the user confirms. A confirmation can happen when the `OK` button is clicked or when `Enter` is pressed.

[onCancel: \(event: DialogInputEvent \) => void](#)

- Required: Only if `isOpen` is not set

The callback that's called when the user cancels. A cancellation can happen when the `Cancel` button is clicked, when the `ESC` key is pressed, or when a click outside of the dialog focus is detected (i.e. in the overlay).

It's not required if `isOpen` is not set (uncontrolled mode), as the component will take care of closing itself, but you can still pass a callback if something must be done upon cancelling (the component will still close itself in this case).

If `isOpen` is set (controlled mode), then it's required, and you need to set the state that defines `isOpen` to `false` as part of this callback if you want the dialog to close when the user cancels.

[confirmButtonText: string](#)

- Required: No
- Default: “OK”

The optional custom text to display as the confirmation button's label

[cancelButtonText: string](#)

- Required: No
- Default: “Cancel”

The optional custom text to display as the cancellation button's label

First published

November 25, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ConfirmDialog”](#)

[Previous ComboboxControl](#) [Previous: ComboboxControl](#)

[Next CustomSelectControlV2](#) [Next: CustomSelectControlV2](#)

CustomSelectControlV2

In this article

[Table of Contents](#)

- [CustomSelect](#)
- [CustomSelectItem](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

[CustomSelect](#)

Used to render a customizable select control component.

Props

The component accepts the following props:

`children: React.ReactNode`

The child elements. This should be composed of `CustomSelect.Item` components.

- Required: yes

`defaultValue: string`

An optional default value for the control. If left `undefined`, the first non-disabled item will be used.

- Required: no

`label: string`

Label for the control.

- Required: yes

`onChange: (newValue: string) => void`

A function that receives the new value of the input.

- Required: no

`renderSelectedValue: (selectValue: string) => React.ReactNode`

Can be used to render select UI with custom styled values.

- Required: no

`size: 'default' | 'large'`

The size of the control.

- Required: no

`value: string`

Can be used to externally control the value of the control.

- Required: no

CustomSelectItem

Used to render a select item.

Props

The component accepts the following props:

`value: string`

The value of the select item. This will be used as the children if children are left `undefined`.

- Required: yes

`children: React.ReactNode`

The children to display for each select item. The `value` will be used if left `undefined`.

- Required: no

First published

November 23, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CustomSelectControlV2"](#)

[Previous ConfirmDialog](#) [Previous: ConfirmDialog](#)
[Next CustomSelectControl](#) [Next: CustomSelectControl](#)

CustomSelectControl

In this article

Table of Contents

- [Design guidelines](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

CustomSelectControl allows users to select an item from a single-option menu just like [SelectControl](#), with the addition of being able to provide custom styles for each item in the menu. This means it does not use a native `<select>`, so should only be used if the custom styling is necessary.

[Design guidelines](#)

These are the same as [the ones for SelectControls](#).

[Development guidelines](#)

[Usage](#)

```
import { useState } from 'react';
import { CustomSelectControl } from '@wordpress/components';

const options = [
  {
    key: 'small',
    name: 'Small',
    style: { fontSize: '50%' },
  },
]
```

```

{
  key: 'normal',
  name: 'Normal',
  style: { fontSize: '100%' },
},
{
  key: 'large',
  name: 'Large',
  style: { fontSize: '200%' },
},
{
  key: 'huge',
  name: 'Huge',
  style: { fontSize: '300%' },
},
];

```

```

function MyCustomSelectControl() {
  const [ , setFontSize ] = useState();
  return (
    <CustomSelectControl
      __nextUnconstrainedWidth
      label="Font Size"
      options={ options }
      onChange={ ( { selectedItem } ) => setFontSize( selectedItem ) }
    />
  );
}

```

```

function MyControlledCustomSelectControl() {
  const [ fontSize, setFontSize ] = useState( options[ 0 ] );
  return (
    <CustomSelectControl
      __nextUnconstrainedWidth
      label="Font Size"
      options={ options }
      onChange={ ( { selectedItem } ) => setFontSize( selectedItem ) }
      value={ options.find( ( option ) => option.key === fontSize.key ) }
    />
  );
}

```

Props

className

A custom class name to append to the outer `<div>`.

- Type: `String`
- Required: No

hideLabelFromVision

Used to visually hide the label. It will always be visible to screen readers.

- Type: Boolean
- Required: No

label

The label for the control.

- Type: String
- Required: Yes

describedBy

Pass in a description that will be shown to screen readers associated with the select trigger button. If no value is passed, the text “Currently selected: selectedItem.name” will be used fully translated.

- Type: String
- Required: No

options

The options that can be chosen from.

- Type: Array<{ key: String, name: String, style: ?{}, className: ?String, ...rest }>
- Required: Yes

onChange

Function called with the control’s internal state changes. The selectedItem property contains the next selected item.

- Type: Function
- Required: No

value

Can be used to externally control the value of the control, like in the MyControlledCustomSelectControl example above.

- Type: Object
- Required: No

__nextUnconstrainedWidth

Start opting into the unconstrained width style that will become the default in a future version, currently scheduled to be WordPress 6.4. (The prop can be safely removed once this happens.)

- Type: Boolean
- Required: No
- Default: `false`

onMouseOver

A handler for onMouseOver events.

- Type: Function
- Required: No

onMouseOut

A handler for onMouseOut events.

- Type: Function
- Required: No

onFocus

A handler for onFocus events.

- Type: Function
- Required: No

onBlur

A handler for onBlur events.

- Type: Function
- Required: No

Related components

- Like this component, but implemented using a native `<select>` for when custom styling is not necessary, the `SelectControl` component.
- To select one option from a set, when you want to show all the available options at once, use the `Radio` component.
- To select one or more items from a set, use the `CheckboxControl` component.
- To toggle a single setting on or off, use the `ToggleControl` component.
- If you have a lot of items, `ComboboxControl` might be a better fit.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: CustomSelectControl”](#)

[Previous CustomSelectControlV2](#) [Previous: CustomSelectControlV2](#)

[Next Dashicon](#) [Next: Dashicon](#)

Dashicon

[↑ Back to top](#)

Usage

```
import { Dashicon } from '@wordpress/components';

const MyDashicon = () => (
  <div>
    <Dashicon icon="admin-home" />
    <Dashicon icon="products" />
    <Dashicon icon="wordpress" />
  </div>
);
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Dashicon”](#)

[Previous CustomSelectControl](#) [Previous: CustomSelectControl](#)

[Next DateTime](#) [Next: DateTime](#)

DateTime

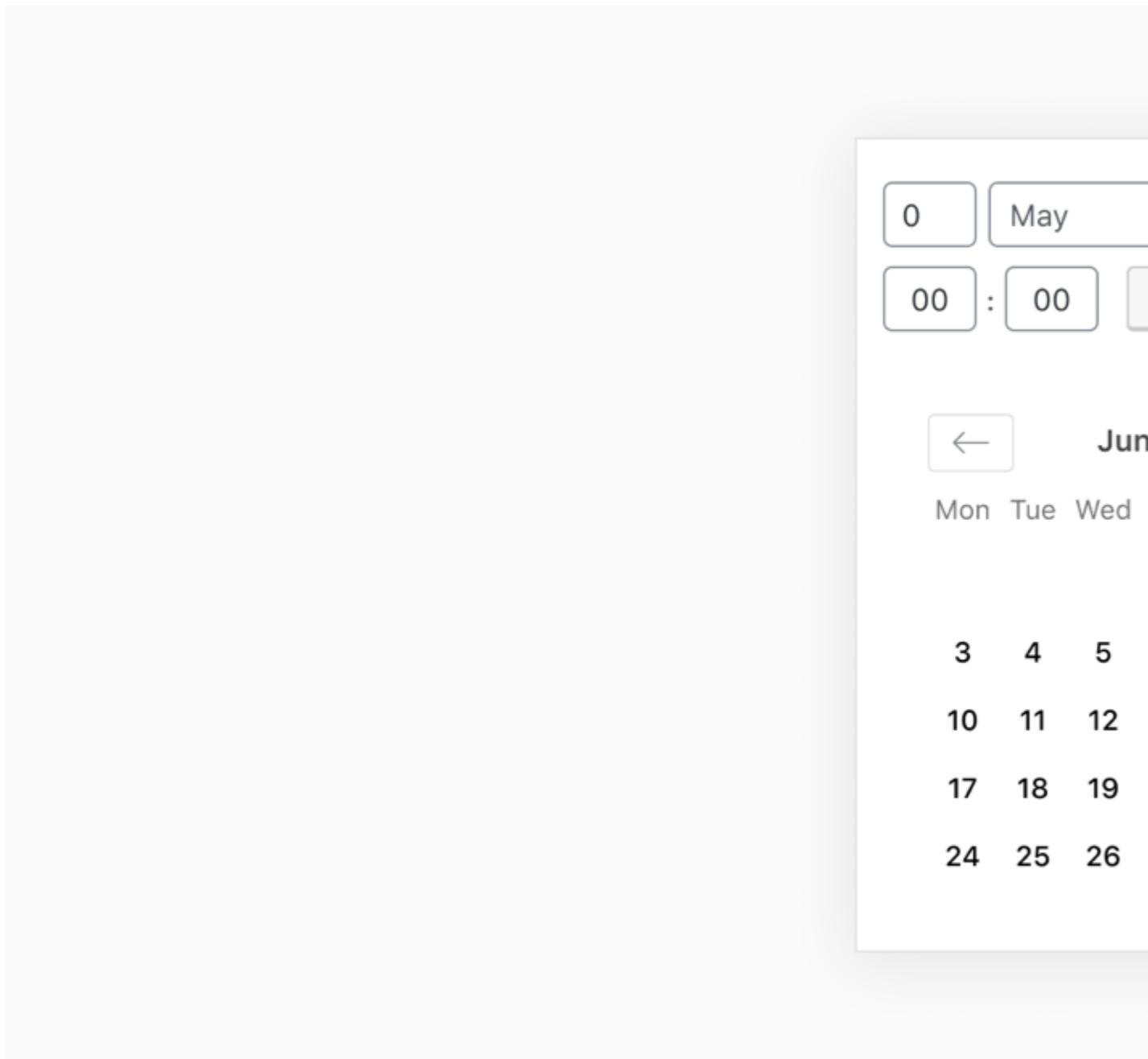
In this article

[Table of Contents](#)

- [Best practices](#)
- [Usage](#)
- [Props](#)
 - [currentDate: Date | string | number | null](#)
 - [onChange: \(date: string | null \) => void](#)
 - [is12Hour: boolean](#)
 - [isValidDate: \(date: Date \) => boolean](#)
 - [onMonthPreviewed: \(date: Date \) => void](#)
 - [events: { date: Date }\[\]](#)
 - [startOfWeek: number](#)

[↑ Back to top](#)

DateTimePicker is a React component that renders a calendar and clock for date and time selection. The calendar and clock components can be accessed individually using the `DatePicker` and `TimePicker` components respectively.



Best practices

Date pickers should:

- Use smart defaults and highlight the current date.

Usage

Render a DateTimePicker.

```
import { useState } from 'react';
import { DateTimePicker } from '@wordpress/components';

const MyDateTimePicker = () => {
  const [ date, setDate ] = useState( new Date() );
```

```

    return (
      <DateTimePicker
        currentDate={ date }
        onChange={ ( newDate ) => setDate( newDate ) }
        is12Hour={ true }
      />
    );
}

```

Props

The component accepts the following props:

currentDate: Date | string | number | null

The current date and time at initialization. Optionally pass in a `null` value to specify no date is currently selected.

- Required: No
- Default: today's date

onChange: (date: string | null) => void

The function called when a new date or time has been selected. It is passed the `currentDate` as an argument.

- Required: No

is12Hour: boolean

Whether we use a 12-hour clock. With a 12-hour clock, an AM/PM widget is displayed and the time format is assumed to be MM-DD-YYYY (as opposed to the default format DD-MM-YYYY).

- Type: `bool`
- Required: No
- Default: false

isValidDate: (date: Date) => boolean

A callback function which receives a Date object representing a day as an argument, and should return a Boolean to signify if the day is valid or not.

- Required: No

onMonthPreviewed: (date: Date) => void

A callback invoked when selecting the previous/next month in the date picker. The callback receives the new month date in the ISO format as an argument.

- Required: No

[events: { date: Date }\[\]](#)

List of events to show in the date picker. Each event will appear as a dot on the day of the event.

- Type: `Array`
- Required: No

[startOfWeek: number](#)

The day that the week should start on. 0 for Sunday, 1 for Monday, etc.

- Required: No
- Default: 0

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: DateTime](#)

[Previous Dashicon](#) [Previous: Dashicon](#)

[Next DimensionControl](#) [Next: DimensionControl](#)

DimensionControl

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [label](#)
 - [value](#)
 - [sizes](#)
 - [icon](#)
 - [onChange](#)
 - [className](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

DimensionControl is a component designed to provide a UI to control spacing and/or dimensions.

Usage

```
import { useState } from 'react';
import { __experimentalDimensionControl as DimensionControl } from '@wordp

export default function MyCustomDimensionControl() {
  const [ paddingSize, setPaddingSize ] = useState( '' );

  return (
    <DimensionControl
      label={ 'Padding' }
      icon={ 'desktop' }
      onChange={ ( value ) => setPaddingSize( value ) }
      value={ paddingSize }
    />
  );
}
```

Note: by default, if you do not provide an initial `value` prop for the current dimension value, then no value will be selected (ie: there is no default dimension set).

Props

label

- **Type:** `string`
- **Required:** Yes

The human readable label for the control.

value

- **Type:** `string`
- **Required:** No

The current value of the dimension UI control. If provided the UI will automatically select the value.

sizes

- **Type:** `{ name: string; slug: string }[]`
- **Default:** See `packages/block-editor/src/components/dimension-control/sizes.ts`
- **Required:** No

An optional array of size objects in the following shape:

```
[  
  {  
    name: __( 'Small' ),  
    slug: 'small',  
  },
```

```
{  
  name: __( 'Medium' ),  
  slug: 'small',  
},  
// ...etc  
]
```

By default a set of relative sizes (small, medium...etc) are provided. See `packages/block-editor/src/components/dimension-control/sizes.js`.

[icon](#)

- **Type:** `string`
- **Required:** No

An optional dashicon to display before to the control label.

[onChange](#)

- **Type:** `(value?: string) => void;`
- **Required:** No
- **Arguments:**
 - `size` – a string representing the selected size (eg: `medium`)

A callback which is triggered when a spacing size value changes (is selected/clicked).

[className](#)

- **Type:** `string`
- **Default:** ''
- **Required:** No

A string of classes to be added to the control component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: DimensionControl](#)”

[Previous](#) [DateTime](#) [Previous: DateTime](#)

[Next](#) [DropdownMenuV2Ariakit](#) [Next: DropdownMenuV2Ariakit](#)

DropdownMenuV2Ariakit

In this article

Table of Contents

- [Design guidelines](#)
 - [Usage](#)
- [Development guidelines](#)
 - [DropdownMenu](#)
 - [DropdownMenuItem](#)
 - [DropdownMenuCheckboxItem](#)
 - [DropdownMenuRadioItem](#)
 - [DropdownMenuItemLabel](#)
 - [DropdownMenuItemHelpText](#)
 - [DropdownMenuGroup](#)
 - [DropdownMenuSeparatorProps](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

DropdownMenu displays a menu to the user (such as a set of actions or functions) triggered by a button.

[Design guidelines](#)

[Usage](#)

When to use a DropdownMenu

Use a DropdownMenu when you want users to:

- Choose an action or change a setting from a list, AND
- Only see the available choices contextually.

DropdownMenu is a React component to render an expandable menu of buttons. It is similar in purpose to a `<select>` element, with the distinction that it does not maintain a value. Instead, each option behaves as an action button.

If you need to display all the available options at all times, consider using a Toolbar instead. Use a DropdownMenu to display a list of actions after the user interacts with a button.

Do

Use a DropdownMenu to display a list of actions after the user interacts with an icon.

Don't use a DropdownMenu for important actions that should always be visible. Use a Toolbar instead.

Don't

Don't use a `DropdownMenu` for frequently used actions. Use a `Toolbar` instead.

Behavior

Generally, the parent button should indicate that interacting with it will show a `DropdownMenu`.

The parent button should retain the same visual styling regardless of whether the `DropdownMenu` is displayed or not.

Placement

The `DropdownMenu` should typically appear directly below, or below and to the left of, the parent button. If there isn't enough space below to display the full `DropdownMenu`, it can be displayed instead above the parent button.

Development guidelines

This component is still highly experimental, and it's not normally accessible to consumers of the `@wordpress/components` package.

The component exposes a set of components that are meant to be used in combination with each other in order to implement a `DropdownMenu` correctly.

DropdownMenu

The root component, used to specify the menu's trigger and its contents.

Props

The component accepts the following props:

`trigger: React.ReactNode`

The trigger button

- Required: yes

`children: React.ReactNode`

The contents of the dropdown

- Required: yes

`defaultOpen: boolean`

The open state of the dropdown menu when it is initially rendered. Use when not wanting to control its open state.

- Required: no
- Default: `false`

`open: boolean`

The controlled open state of the dropdown menu. Must be used in conjunction with `onOpenChange`.

- Required: no

`onOpenChange: (open: boolean) => void`

Event handler called when the open state of the dropdown menu changes.

- Required: no

`modal: boolean`

The modality of the dropdown menu. When set to true, interaction with outside elements will be disabled and only menu content will be visible to screen readers.

- Required: no
- Default: `true`

`placement: "top" | "top-start" | "top-end" | "right" | "right-start" | "right-end" | "bottom" | "bottom-start" | "bottom-end" | "left" | "left-start" | "left-end"`

The placement of the dropdown menu popover.

- Required: no
- Default: '`bottom-start`' for root-level menus, '`right-start`' for nested menus

`gutter: number`

The distance in pixels from the trigger.

- Required: no
- Default: 8 for root-level menus, 16 for nested menus

`shift: number`

The skidding of the popover along the anchor element. Can be set to negative values to make the popover shift to the opposite side.

- Required: no
- Default: 0 for root-level menus, -8 for nested menus

[DropdownMenuItem](#)

Used to render a menu item.

Props

The component accepts the following props:

`children: React.ReactNode`

The contents of the item

- Required: yes

`prefix: React.ReactNode`

The contents of the item's prefix.

- Required: no

`suffix: React.ReactNode`

The contents of the item's suffix.

- Required: no

`hideOnClick: boolean`

Whether to hide the dropdown menu when the menu item is clicked.

- Required: no
- Default: `true`

`disabled: boolean`

Determines if the element is disabled.

- Required: no
- Default: `false`

[DropdownMenuItem](#)

Used to render a checkbox item.

Props

The component accepts the following props:

`children: React.ReactNode`

The contents of the item

- Required: yes

`suffix: React.ReactNode`

The contents of the item's suffix.

- Required: no

`hideOnClick: boolean`

Whether to hide the dropdown menu when the menu item is clicked.

- Required: no
- Default: `false`

`disabled: boolean`

Determines if the element is disabled.

- Required: no
- Default: `false`

`name: string`

The checkbox item's name.

- Required: yes

`value: string`

The checkbox item's value, useful when using multiple checkbox items associated to the same name.

- Required: no

`checked: boolean`

The checkbox item's value, useful when using multiple checkbox items associated to the same name.

- Required: no

`defaultChecked: boolean`

The checked state of the checkbox menu item when it is initially rendered. Use when not wanting to control its checked state.

- Required: no

`onChange: (event: React.ChangeEvent< HTMLInputElement >) => void;`

Event handler called when the checked state of the checkbox menu item changes.

- Required: no

[DropdownMenuItem](#)

Used to render a radio item.

Props

The component accepts the following props:

`children: React.ReactNode`

The contents of the item

- Required: yes

`suffix: React.ReactNode`

The contents of the item's suffix.

- Required: no

`hideOnClick: boolean`

Whether to hide the dropdown menu when the menu item is clicked.

- Required: no
- Default: `false`

`disabled: boolean`

Determines if the element is disabled.

- Required: no
- Default: `false`

`name: string`

The radio item's name.

- Required: yes

`value: string | number`

The radio item's value.

- Required: yes

`checked: boolean`

The checkbox item's value, useful when using multiple checkbox items associated to the same name.

- Required: no

`defaultChecked: boolean`

The checked state of the radio menu item when it is initially rendered. Use when not wanting to control its checked state.

- Required: no

`onChange: (event: React.ChangeEvent< HTMLInputElement >) => void;`

Event handler called when the checked radio menu item changes.

- Required: no

[DropdownMenuItemLabel](#)

Used to render the menu item's label.

Props

The component accepts the following props:

`children: React.ReactNode`

The label contents.

- Required: yes

[DropdownMenuItemLabelText](#)

Used to render the menu item's help text.

Props

The component accepts the following props:

`children: React.ReactNode`

The help text contents.

- Required: yes

[DropdownMenuGroup](#)

Used to group menu items.

Props

The component accepts the following props:

children: React.ReactNode

The contents of the group.

- Required: yes

[**DropdownMenuSeparatorProps**](#)

Used to render a visual separator.

First published

October 25, 2023

Last updated

December 22, 2023

Edit article

[Improve it on GitHub: DropdownMenuV2Ariakit”](#)

[Previous DimensionControl](#) [Previous: DimensionControl](#)

[Next Disabled](#) [Next: Disabled](#)

Disabled

In this article

[Table of Contents](#)

- [Usage](#)
 - [Props](#)

[↑ Back to top](#)

Disabled is a component which disables descendant tabbable elements and prevents pointer interaction.

[**Usage**](#)

Assuming you have a form component, you can disable all form inputs by wrapping the form with <Disabled>.

```
import { useState } from 'react';
import { Button, Disabled, TextControl } from '@wordpress/components';

const MyDisabled = () => {
    const [ isEnabled, setIsEnabled ] = useState( true );

    let input = <TextControl label="Input" onChange={ () => {} } />;
```

```

    if ( isEnabled ) {
      input = <Disabled>{ input }</Disabled>;
    }

    const toggleDisabled = () => {
      setIsDisabled( ( state ) => ! state );
    };

    return (
      <div>
        { input }
        <Button variant="primary" onClick={ toggleDisabled }>
          Toggle Disabled
        </Button>
      </div>
    );
  };
}

```

A component can detect if it has been wrapped in a `<Disabled />` by accessing its [context](#) using `Disabled.Context`.

```

function CustomButton( props ) {
  const isEnabled = useContext( Disabled.Context );
  return (
    <button
      { ...props }
      style={ { opacity: isEnabled ? 0.5 : 1 } }
    />
  );
}

```

Note: this component may not behave as expected in browsers that don't support [the inert HTML attribute](#). We recommend adding [the official WICG polyfill](#) when using this component in your project.

[Props](#)

The component accepts the following props:

isDisabled

Whether to disable all the descendant fields. Defaults to `true`.

- Type: `Boolean`
- Required: No
- Default: `true`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Disabled”](#)

[Previous DropdownMenuV2Ariakit](#) [Previous: DropdownMenuV2Ariakit](#)

[Next Divider](#) [Next: Divider](#)

Divider

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [margin: number](#)
 - [marginEnd: number](#)
 - [marginStart: number](#)
 - [orientation: horizontal | vertical](#)
 - [Inherited props](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`Divider` is a layout component that separates groups of related content.

[Usage](#)

```
import {  
    __experimentalDivider as Divider,  
    __experimentalText as Text,  
    __experimentalVStack as VStack,  
} from `@wordpress/components`;  
  
function Example() {  
    return (  
        <VStack spacing={4}>  
            <Text>Some text here</Text>  
            <Divider />  
            <Text>Some more text here</Text>  
        </VStack>  
    );  
}
```

Props

margin: number

Adjusts all margins on the inline dimension.

- Required: No

marginEnd: number

Adjusts the inline-end margin.

- Required: No

marginStart: number

Adjusts the inline-start margin.

- Required: No

orientation: horizontal | vertical

Divider's orientation. When using inside a flex container, you may need to make sure the divider is `stretch` aligned in order for it to be visible.

- Required: No
- Default: `horizontal`

Inherited props

Divider also inherits all of the [Separator props](#).

First published

May 7, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Divider](#)”

[Previous](#) [Disabled](#) [Previous: Disabled](#)
[Next](#) [Draggable](#) [Next: Draggable](#)

Draggable

In this article

Table of Contents

- [Props](#)
 - [appendToOwnerDocument: boolean](#)
 - [elementId: string](#)
 - [onDragEnd: \(event: DragEvent \) => void](#)
 - [onDragOver: \(event: DragEvent \) => void](#)
 - [onDragStart: \(event: DragEvent \) => void](#)
 - [transferData: unknown](#)
- [Usage](#)

[↑ Back to top](#)

Draggable is a Component that provides a way to set up a cross-browser (including IE) customizable drag image and the transfer data for the drag event. It decouples the drag handle and the element to drag: use it by wrapping the component that will become the drag handle and providing the DOM ID of the element to drag.

Note that the drag handle needs to declare the `draggable="true"` property and bind the Draggables `onDraggableStart` and `onDraggableEnd` event handlers to its own `onDragStart` and `onDragEnd` respectively. **Draggable** takes care of the logic to setup the drag image and the transfer data, but is not concerned with creating an actual DOM element that is draggable.

[Props](#)

The component accepts the following props:

[**appendToOwnerDocument: boolean**](#)

Whether to append the cloned element to the `ownerDocument` body. By default, elements sourced by id are appended to the element's wrapper.

- Required: No
- Default: `false`

[**elementId: string**](#)

The HTML id of the element to clone on drag.

- Required: Yes

onDragEnd: (event: DragEvent)=> void

A function called when dragging ends. This callback receives the `event` object from the `dragend` event as its first parameter.

- Required: No
- Default: `noop`

onDragOver: (event: DragEvent)=> void

A function called when the element being dragged is dragged over a valid drop target. This callback receives the `event` object from the `dragover` event as its first parameter.

- Required: No
- Default: `noop`

onDragStart: (event: DragEvent)=> void

A function called when dragging starts. This callback receives the `event` object from the `dragstart` event as its first parameter.

- Required: No
- Default: `noop`

transferData: unknown

Arbitrary data object attached to the drag and drop event.

- Required: Yes

Usage

```
import { Draggable, Panel, PanelBody } from '@wordpress/components';
import { Icon, more } from '@wordpress/icons';

const MyDraggable = () => (
    <div id="draggable-panel">
        <Panel header="Draggable panel">
            <PanelBody>
                <Draggable elementId="draggable-panel" transferData={ {} }>
                    { ( { onDraggableStart, onDraggableEnd } ) => (
                        <div
                            className="example-drag-handle"
                            draggable
                            onDragStart={ onDraggableStart }
                            onDragEnd={ onDraggableEnd }
                        >
                            <Icon icon={ more } />
                        </div>
                    ) }
                </Draggable>
            </PanelBody>
        </Panel>
    </div>
)
```

```
        </Panel>
    </div>
);
```

In case you want to call your own `dragstart` / `dragend` event handlers as well, you can pass them to `Draggable` and it'll take care of calling them after their own:

```
import { Draggable, Panel, PanelBody } from '@wordpress/components';
import { Icon, more } from '@wordpress/icons';

const MyDraggable = ( { onDragStart, onDragEnd } ) => (
    <div id="draggable-panel">
        <Panel header="Draggable panel">
            <PanelBody>
                <Draggable
                    elementId="draggable-panel"
                    transferData={ {} }
                    onDragStart={ onDragStart }
                    onDragEnd={ onDragEnd }
                >
                    { ( { onDraggableStart, onDraggableEnd } ) => (
                        <div
                            className="example-drag-handle"
                            draggable
                            onDragStart={ onDraggableStart }
                            onDragEnd={ onDraggableEnd }
                        >
                            <Icon icon={ more } />
                        </div>
                    ) }
                </Draggable>
            </PanelBody>
        </Panel>
    </div>
);
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Draggable](#)

[Previous Divider](#) [Previous: Divider](#)
[Next DropZone](#) [Next: DropZone](#)

DropZone

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [className](#)
 - [label](#)
 - [onFilesDrop](#)
 - [onHTMLDrop](#)
 - [onDrop](#)

[↑ Back to top](#)

DropZone is a component creating a drop zone area taking the full size of its parent element. It supports dropping files, HTML content or any other HTML drop event.

[Usage](#)

```
import { useState } from 'react';
import { DropZone } from '@wordpress/components';

const MyDropZone = () => {
    const [ hasDropped, setHasDropped ] = useState( false );

    return (
        <div>
            { hasDropped ? 'Dropped!' : 'Drop something here' }
            <DropZone
                onFilesDrop={ () => setHasDropped( true ) }
                onHTMLDrop={ () => setHasDropped( true ) }
                onDrop={ () => setHasDropped( true ) }
            />
        </div>
    );
}
```

[Props](#)

The component accepts the following props:

[className](#)

A CSS `class` to give to the wrapper element.

- Type: `String`
- Default: `undefined`

[label](#)

A string to be shown within the drop zone area.

- Type: `String`
- Default: `Drop files to upload`

[onFilesDrop](#)

The function is called when dropping a file into the `DropZone`. It receives an array of dropped files as an argument.

- Type: `Function`
- Required: No
- Default: `noop`

[onHTMLDrop](#)

The function is called when dropping HTML into the `DropZone`. It receives the HTML being dropped as an argument.

- Type: `Function`
- Required: No
- Default: `noop`

[onDrop](#)

The function is generic drop handler called if the `onFilesDrop` or `onHTMLDrop` are not called. It receives the drop `event` object as an argument.

- Type: `Function`
- Required: No
- Default: `noop`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: DropZone](#)”

[Previous Draggable](#) [Previous: Draggable](#)
[Next DropdownMenu](#) [Next: DropdownMenu](#)

DropdownMenu

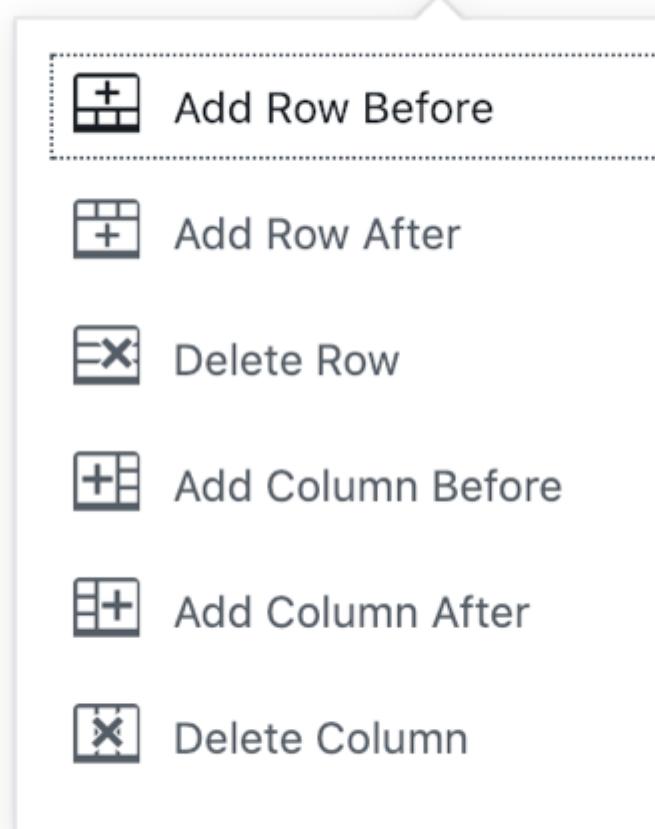
In this article

Table of Contents

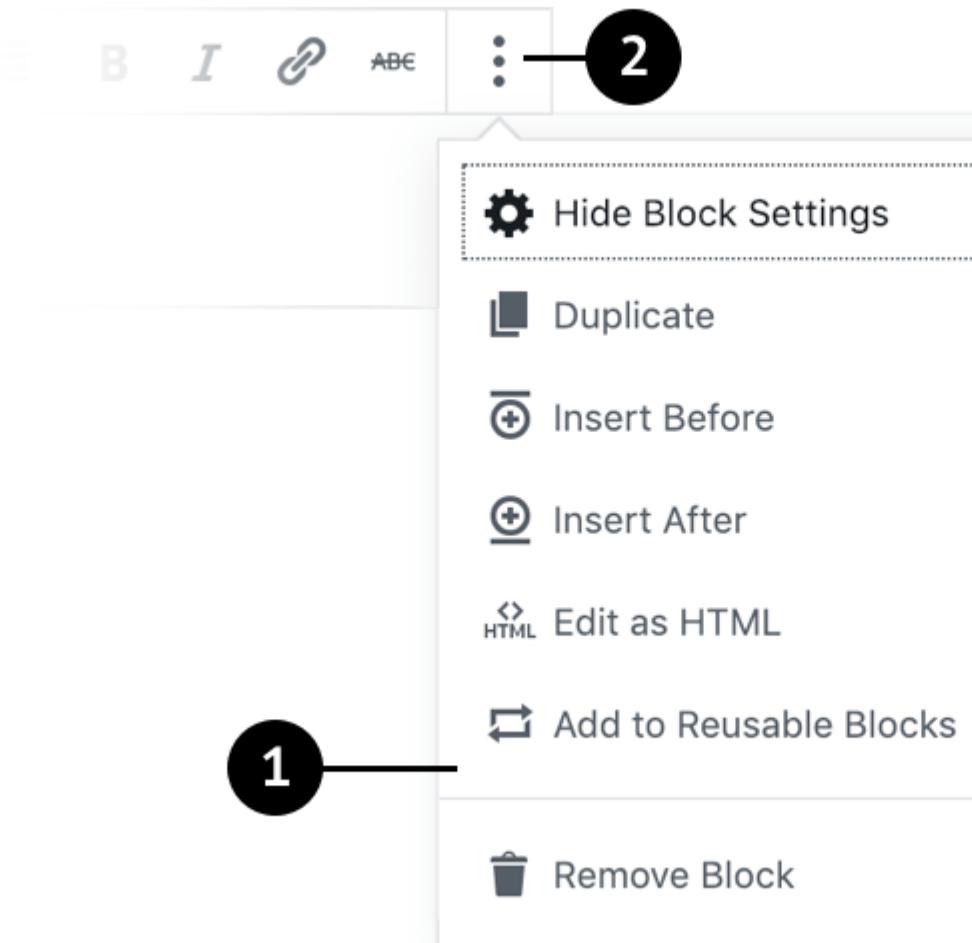
- [Anatomy](#)
- [Design guidelines](#)
 - [Usage](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
 - [defaultOpen: boolean](#)
 - [open: boolean](#)
 - [onToggle: \(willOpen: boolean \) => void](#)

[↑ Back to top](#)

The DropdownMenu displays a list of actions (each contained in a MenuItem, MenuItemChoice, or MenuGroup) in a compact way. It appears in a Popover after the user has interacted with an element (a button or icon) or when they perform a specific action.



Anatomy



1. Popover: a container component in which the DropdownMenu is wrapped.
2. Parent button: the icon or button that is used to toggle the display of the Popover containing the DropdownMenu.
3. MenuItem: the list items within the DropdownMenu.

Design guidelines

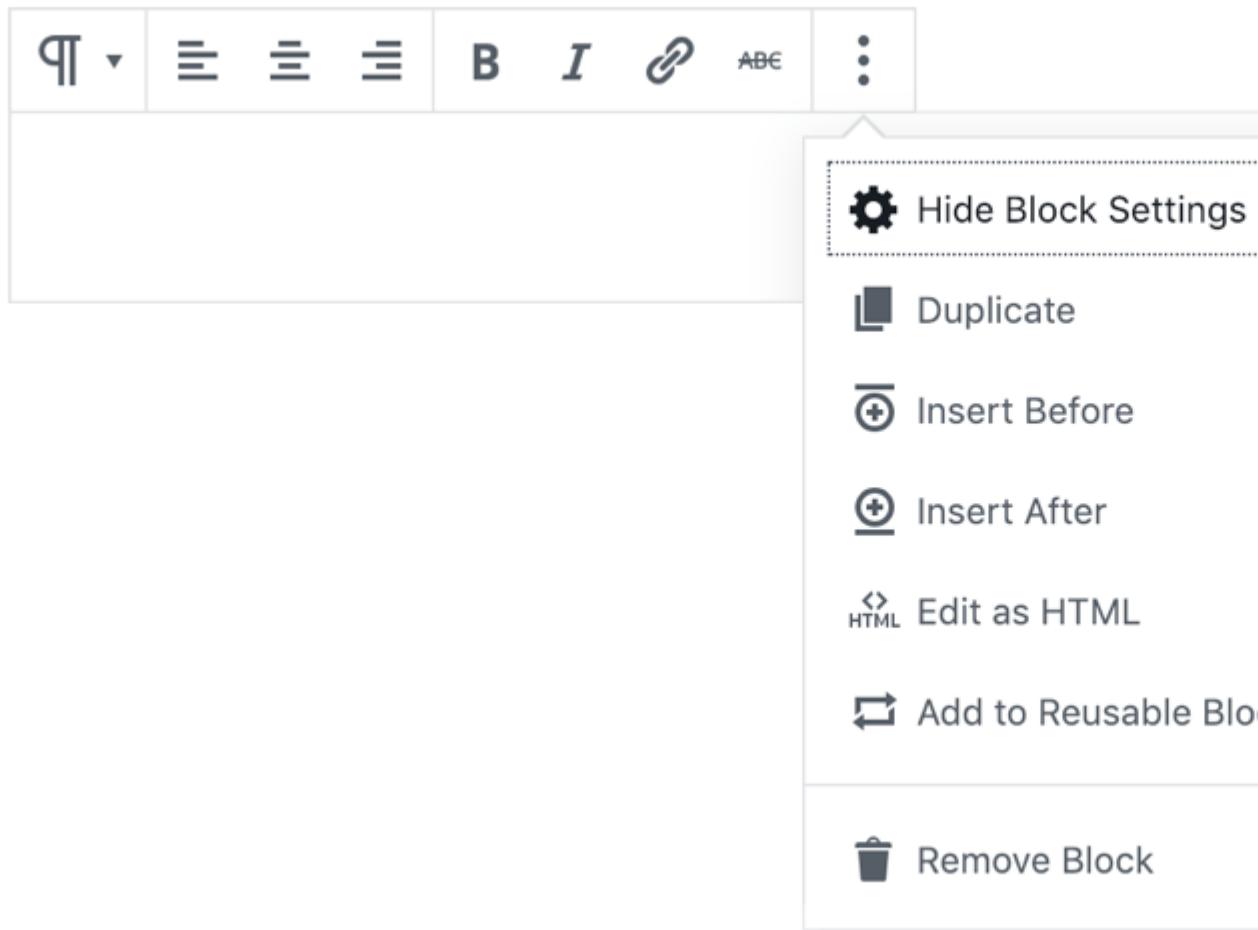
Usage

When to use a DropdownMenu

Use a DropdownMenu when you want users to:

- Choose an action or change a setting from a list, AND
- Only see the available choices contextually.

If you need to display all the available options at all times, consider using a Toolbar instead.



Do

Use a `DropdownMenu` to display a list of actions after the user interacts with an icon.



Left Align

Center Align

Right Align

Bold

Italic

Link

Strikethrough

Don't

Don't use a DropdownMenu for frequently used actions. Use a Toolbar instead.

Behavior

Generally, the parent button should have a triangular icon to the right of the icon or text to indicate that interacting with it will show a DropdownMenu. In rare cases where the parent button directly indicates that there'll be more content (through the use of an ellipsis or "More" label), this can be omitted.

The parent button should retain the same visual styling regardless of whether the DropdownMenu is displayed or not.

Placement

The DropdownMenu should typically appear directly below, or below and to the left of, the parent button. If there isn't enough space below to display the full DropdownMenu, it can be displayed instead above the parent button.

Development guidelines

DropdownMenu is a React component to render an expandable menu of buttons. It is similar in purpose to a `<select>` element, with the distinction that it does not maintain a value. Instead, each option behaves as an action button.

Usage

Render a Dropdown Menu with a set of controls:

```
import { DropdownMenu } from '@wordpress/components';
import {
    more,
    arrowLeft,
    arrowRight,
    arrowUp,
    arrowDown,
} from '@wordpress/icons';

const MyDropdownMenu = () => (
    <DropdownMenu
        icon={ more }
        label="Select a direction"
        controls={[
            {
                title: 'Up',
                icon: arrowUp,
                onClick: () => console.log( 'up' ),
            },
            {
                title: 'Right',
                icon: arrowRight,
                onClick: () => console.log( 'right' ),
            },
            {
                title: 'Down',
                icon: arrowDown,
                onClick: () => console.log( 'down' ),
            },
            {
                title: 'Left',
                icon: arrowLeft,
                onClick: () => console.log( 'left' ),
            },
        ] }
    />
);
```

Alternatively, specify a `children` function which returns elements valid for use in a `DropdownMenu`: `MenuItem`, `MenuItemChoice`, or `MenuGroup`.

```
import { DropdownMenu, MenuGroup, MenuItem } from '@wordpress/components';
import { more, arrowUp, arrowDown, trash } from '@wordpress/icons';
```

```

const MyDropdownMenu = () => (
  <DropdownMenu icon={{ more }} label="Select a direction">
    { ( { onClose } ) => (
      <>
        <MenuGroup>
          <MenuItem icon={{ arrowUp }} onClick={{ onClose }}>
            Move Up
          </MenuItem>
          <MenuItem icon={{ arrowDown }} onClick={{ onClose }}>
            Move Down
          </MenuItem>
        </MenuGroup>
        <MenuGroup>
          <MenuItem icon={{ trash }} onClick={{ onClose }}>
            Remove
          </MenuItem>
        </MenuGroup>
      </>
    ) }
  </DropdownMenu>
);

```

Props

The component accepts the following props:

`icon: string | null`

The [Dashicon](#) icon slug to be shown in the collapsed menu button.

- Required: No
- Default: "menu"

See also: <https://developer.wordpress.org/resource/dashicons/>

`label: string`

A human-readable label to present as accessibility text on the focused collapsed menu button.

- Required: Yes

`controls: DropdownOption[] | DropdownOption[][]`

An array or nested array of objects describing the options to be shown in the expanded menu.

Each object should include an `icon` [Dashicon](#) slug string, a human-readable `title` string, `isDisabled` boolean flag and an `onClick` function callback to invoke when the option is selected.

A valid `DropdownMenu` must specify a `controls` or `children` prop, or both.

- Required: No

`children: (callbackProps: DropdownCallbackProps) => ReactNode`

A [function render prop](#) which should return an element or elements valid for use in a `DropdownMenu`: `MenuItem`, `MenuItemChoice`, or `MenuGroup`. Its first argument is a `props` object including the same values as given to a [Dropdown's renderContent](#) (`isOpen`, `onToggle`, `onClose`).

A valid `DropdownMenu` must specify a `controls` or `children` prop, or both.

- Required: No

See also: <https://developer.wordpress.org/resource/dashicons/>

`className: string`

A class name to apply to the dropdown menu's toggle element wrapper.

- Required: No

`popoverProps: DropdownProps['popoverProps']`

Properties of `popoverProps` object will be passed as props to the nested `Popover` component.

Use this object to modify props available for the `Popover` component that are not already exposed in the `DropdownMenu` component, e.g.: the direction in which the popover should open relative to its parent node set with `position` prop.

- Required: No

`toggleProps: ToggleProps`

Properties of `toggleProps` object will be passed as props to the nested `Button` component in the `renderToggle` implementation of the `Dropdown` component used internally.

Use this object to modify props available for the `Button` component that are not already exposed in the `DropdownMenu` component, e.g.: the tooltip text displayed on hover set with `tooltip` prop.

- Required: No

`menuProps: NavigableContainerProps`

Properties of `menuProps` object will be passed as props to the nested `NavigableMenu` component in the `renderContent` implementation of the `Dropdown` component used internally.

Use this object to modify props available for the `NavigableMenu` component that are not already exposed in the `DropdownMenu` component, e.g.: the orientation of the menu set with `orientation` prop.

- Required: No

disableOpenOnArrowDown: boolean

In some contexts, the arrow down key used to open the dropdown menu might need to be disabled—for example when that key is used to perform another action.

- Required: No
- Default: `false`

defaultOpen: boolean

The open state of the dropdown menu when initially rendered. Use when you do not need to control its open state. It will be overridden by the `open` prop if it is specified on the component's first render.

- Required: No

open: boolean

The controlled open state of the dropdown menu. Must be used in conjunction with `onToggle`.

- Required: No

onToggle: (willOpen: boolean) => void

A callback invoked when the state of the dropdown changes from open to closed and vice versa.

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: DropdownMenu](#)

[Previous DropZone](#) [Previous: DropZone](#)
[Next Dropdown](#) [Next: Dropdown](#)

Dropdown

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [className: string](#)

- [contentClassName: string](#)
- [defaultOpen: boolean](#)
- [expandOnMobile: boolean](#)
- [focusOnMount: 'firstElement' | boolean](#)
- [headerTitle: string](#)
- [onClose: \(\) => void](#)
- [open: boolean](#)
- [onToggle: \(willOpen: boolean \) => void](#)
- [popoverProps: WordPressComponentProps< Omit< PopoverProps, 'children' > 'div', false >](#)
- [renderContent: \(props: CallbackProps \) => ReactNode](#)
- [renderToggle: \(props: CallbackProps \) => ReactNode](#)
- [style: React.CSSProperties](#)

[↑ Back to top](#)

Dropdown is a React component to render a button that opens a floating content modal when clicked.

This component takes care of updating the state of the dropdown menu (opened/closed), handles closing the menu when clicking outside and uses render props to render the button and the content.

Usage

```
import { Button, Dropdown } from '@wordpress/components';

const MyDropdown = () => (
  <Dropdown
    className="my-container-class-name"
    contentClassName="my-popover-content-classname"
    popoverProps={ { placement: 'bottom-start' } }
    renderToggle={ ( { isOpen, onToggle } ) => (
      <Button
        variant="primary"
        onClick={ onToggle }
        aria-expanded={ isOpen }
      >
        Toggle Popover!
      </Button>
    ) }
    renderContent={ () => <div>This is the content of the popover.</div> }
  />
);
```

Props

The component accepts the following props. Props not included in this set will be applied to the element wrapping Popover content.

className: string

className of the global container

- Required: No

contentClassName: string

If you want to target the dropdown menu for styling purposes, you need to provide a contentClassName because it's not being rendered as a child of the container node.

- Required: No

defaultOpen: boolean

The open state of the dropdown when initially rendered. Use when you do not need to control its open state. It will be overridden by the open prop if it is specified on the component's first render.

- Required: No

expandOnMobile: boolean

Opt-in prop to show popovers fullscreen on mobile.

- Required: No
- Default: false

focusOnMount: 'firstElement' | boolean

By default, the *first tabbable element* in the popover will receive focus when it mounts. This is the same as setting this prop to "firstElement".

Specifying a true value will focus the container instead.

Specifying a false value disables the focus handling entirely (this should only be done when an appropriately accessible substitute behavior exists).

- Required: No
- Default: "firstElement"

headerTitle: string

Set this to customize the text that is shown in the dropdown's header when it is fullscreen on mobile.

- Required: No

onClose: () => void

A callback invoked when the popover should be closed.

- Required: No

[open: boolean](#)

The controlled open state of the dropdown. Must be used in conjunction with `onToggle`.

- Required: No

[onToggle: \(willOpen: boolean \) => void](#)

A callback invoked when the state of the dropdown changes from open to closed and vice versa.

- Required: No

[popoverProps: WordPressComponentProps< Omit< PopoverProps, 'children' > 'div', false >](#)

Properties of `popoverProps` object will be passed as props to the `Popover` component.

Use this object to access properties/features of the `Popover` component that are not already exposed in the `Dropdown` component, e.g.: the ability to have the popover without an arrow.

- Required: No

[renderContent: \(props: CallbackProps \) => ReactNode](#)

A callback invoked to render the content of the dropdown menu.

- `isOpen`: whether the dropdown menu is opened or not
 - `onToggle`: A function switching the dropdown menu's state from open to closed and vice versa
 - `onClose`: A function that closes the menu if invoked
- Required: Yes

[renderToggle: \(props: CallbackProps \) => ReactNode](#)

A callback invoked to render the Dropdown Toggle Button.

Its props are the same as the `renderContent` props.

- Required: Yes

[style: React.CSSProperties](#)

The style of the global container

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Dropdown](#)

[Previous DropdownMenu](#) [Previous: DropdownMenu](#)

[Next DuotonePicker](#) [Next: DuotonePicker](#)

DuotonePicker

In this article

[Table of Contents](#)

- [Usage](#)
- [DuotonePicker Props](#)
 - [colorPalette](#)
 - [duotonePalette](#)
 - [value](#)
 - [onChange](#)
 - [asButtons: boolean](#)
 - [loop: boolean](#)
- [DuotoneSwatch Props](#)
 - [values](#)

[↑ Back to top](#)

Usage

```
import { useState } from 'react';
import { DuotonePicker, DuotoneSwatch } from '@wordpress/components';

const DUOTONE_PALETTE = [
    { colors: [ '#8c00b7', '#fcff41' ], name: 'Purple and yellow', slug: 'purple-and-yellow' },
    { colors: [ '#000097', '#ff4747' ], name: 'Blue and red', slug: 'blue-and-red' }
];

const COLOR_PALETTE = [
    { color: '#ff4747', name: 'Red', slug: 'red' },
    { color: '#fcff41', name: 'Yellow', slug: 'yellow' },
    { color: '#000097', name: 'Blue', slug: 'blue' },
    { color: '#8c00b7', name: 'Purple', slug: 'purple' }
];

const Example = () => {
    const [ duotone, setDuotone ] = useState( [ '#000000', '#ffffff' ] );
    return (
        <>
            <DuotonePicker
                duotonePalette={ DUOTONE_PALETTE }
                colorPalette={ COLOR_PALETTE }
            >

```

```
        value={ duotone }
        onChange={ setDuotone }
      />
    <DuotoneSwatch values={ duotone } />
  </>
);
};
```

DuotonePicker Props

colorPalette

- Type: `Object[]`
- Required: Yes

Array of color presets of the form { `color: '#000000'`, `name: 'Black'`, `slug: 'black'` }.

duotonePalette

- Type: `Object[]`
- Required: Yes

Array of duotone presets of the form { `colors: ['#000000', '#ffffff']`, `name: 'Grayscale'`, `slug: 'grayscale'` }.

value

- Type: `string[]`
- Required: Yes

An array of colors for the duotone effect.

onChange

- Type: `Function`
- Required: Yes

Callback which is called when the duotone colors change.

asButtons: boolean

Whether the control should present as a set of buttons, each with its own tab stop.

- Required: No
- Default: `false`

loop: boolean

Prevents keyboard interaction from wrapping around. Only used when `asButtons` is not true.

- Required: No

- Default: true

DuotoneSwatch Props

values

- Type: string[] | null
- Required: No

An array of colors to show or null to show the placeholder swatch icon.

First published

April 22, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: DuotonePicker”](#)

[Previous Dropdown](#) [Previous: Dropdown](#)

[Next Elevation](#) [Next: Elevation](#)

Elevation

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [active: number](#)
 - [borderRadius: CSSProperties\[‘borderRadius’ \]](#)
 - [focus: number](#)
 - [hover: number](#)
 - [isInteractive: boolean](#)
 - [offset: number](#)
 - [value: number](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Elevation is a core component that renders shadow, using the component system’s shadow system.

Usage

The shadow effect is generated using the `value` prop.

```
import {
  __experimentalElevation as Elevation,
  __experimentalSurface as Surface,
  __experimentalText as Text,
} from '@wordpress/components';

function Example() {
  return (
    <Surface>
      <Text>Code is Poetry</Text>
      <Elevation value={ 5 } />
    </Surface>
  );
}
```

Props

active: number

Size of the shadow value when active (see the `value` and `isInteractive` props).

- Required: No

borderRadius: CSSProperties[‘borderRadius’]

Renders the border-radius of the shadow.

- Required: No
- Default: `inherit`

focus: number

Size of the shadow value when focused (see the `value` and `isInteractive` props).

- Required: No

hover: number

Size of the shadow value when hovered (see the `value` and `isInteractive` props).

- Required: No

isInteractive: boolean

Determines if `hover`, `active`, and `focus` shadow values should be automatically calculated and rendered.

- Required: No

- Default: false

[offset: number](#)

Dimensional offsets (margin) for the shadow.

- Required: No
- Default: 0

[value: number](#)

Size of the shadow, based on the Style system's elevation system. The value determines the strength of the shadow, which sense of depth.

In the example below, `isInteractive` is activated to give a better sense of depth.

```
import { __experimentalElevation as Elevation } from '@wordpress/component'

function Example() {
    return (
        <div>
            <Elevation isInteractive value={ 200 } />
        </div>
    );
}
```

- Required: No
- Default: 0

First published

May 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Elevation”](#)

[Previous DuotonePicker](#) [Previous: DuotonePicker](#)
[Next ExternalLink](#) [Next: ExternalLink](#)

ExternalLink

In this article

Table of Contents

- [Usage](#)

- [Props](#)
 - [children: ReactNode](#)
 - [href: string](#)

[↑ Back to top](#)

Link to an external resource.

[Usage](#)

```
import { ExternalLink } from '@wordpress/components';

const MyExternalLink = () => (
  <ExternalLink href="https://wordpress.org">WordPress.org</ExternalLink>
);
```

[Props](#)

The component accepts the following props. Any other props will be passed through to the a.

[children: ReactNode](#)

The content to be displayed within the link.

- Required: Yes

[href: string](#)

The URL of the external resource.

- Required: Yes

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ExternalLink](#)

[Previous Elevation](#) [Previous: Elevation](#)

[Next Flyout](#) [Next: Flyout](#)

Flyout

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [state: PopoverStateReturn](#)
 - [label: string](#)
 - [animated: boolean](#)
 - [animationDuration: boolean](#)
 - [baseId: string](#)
 - [elevation: number](#)
 - [maxWidth: CSSProperties\[‘maxWidth’ \]](#)
 - [onVisibleChange: \(...args: any \) => void](#)
 - [trigger: FunctionComponentElement< any >](#)
 - [visible: boolean](#)
 - [placement: PopperPlacement](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Flyout is a component to render a floating content modal. It is similar in purpose to a tooltip, but renders content of any sort, not only simple text.

[Usage](#)

```
import { Button, __experimentalFlyout as Flyout, __experimentalText as Text } from 'react-native';

function Example() {
  return (
    <Flyout trigger={<Button>Show/Hide content</Button>}>
      <Text>Code is Poetry</Text>
    </Flyout>
  );
}
```

[Props](#)

[state: PopoverStateReturn](#)

- Required: No

[label: string](#)

- Required: No

[animated: boolean](#)

Determines if Flyout has animations.

- Required: No
- Default: true

[animationDuration: boolean](#)

The duration of Flyout animations.

- Required: No
- Default: 160

[baseId: string](#)

ID that will serve as a base for all the items IDs. See <https://reakit.io/docs/popover/#usepopoverstate>

- Required: No
- Default: 160

[elevation: number](#)

Size of the elevation shadow. For more information, check out [Card](#).

- Required: No
- Default: 5

[maxWidth: CSSProperties\[‘maxWidth’ \]](#)

Max-width for the Flyout element.

- Required: No
- Default: 360

[onVisibleChange: \(...args: any \) => void](#)

Callback for when the visible state changes.

- Required: No

[trigger: FunctionComponentElement< any >](#)

Element that triggers the visible state of Flyout when clicked.

```
<Flyout trigger={<Button>Greet</Button>}>
  <Text>Hi! I'm Olaf!</Text>
</Flyout>
```

- Required: Yes

[visible: boolean](#)

Whether Flyout is visible. See [the Reakit docs](#) for more information.

- Required: No
- Default: `false`

[placement: PopperPlacement](#)

Position of the popover element. See [the popper docs](#) for more information.

- Required: No
- Default: `auto`

First published

July 6, 2021

Last updated

May 18, 2022

Edit article

[Improve it on GitHub: Flyout”](#)

[Previous ExternalLink Previous: ExternalLink](#)
[Next FlexBlock Next: FlexBlock](#)

FlexBlock

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [display: CSSProperties\['display'\]](#)

[↑ Back to top](#)

A layout component to contain items of a fixed width within Flex.

[Usage](#)

See [flex/README.md#usage](#) for how to use FlexBlock.

Props

display: CSSProperties['display']

The CSS display property of FlexBlock.

- Required: No

First published

April 30, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FlexBlock](#)”

[Previous Flyout Previous: Flyout](#)

[Next FlexItem Next: FlexItem](#)

FlexItem

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [display: CSSProperties\['display'\]](#)
 - [isBlock: boolean](#)

[↑ Back to top](#)

A layout component to contain items of a fixed width within Flex.

Usage

See [flex/README.md#usage](#) for how to use FlexItem.

Props

display: CSSProperties['display']

The CSS display property of FlexItem.

- Required: No

[isBlock: boolean](#)

Determines if `FlexItem` should render as an adaptive full-width block.

- Required: No
- Default: `false`

First published

April 30, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FlexItem”](#)

[Previous FlexBlock](#) [Previous: FlexBlock](#)

[Next Flex](#) [Next: Flex](#)

Flex

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [align: CSSProperties\['alignItems'\]](#)
 - [direction: ResponsiveCSSValue<CSSProperties\['flexDirection'\]>](#)
 - [expanded: boolean](#)
 - [gap: number](#)

[↑ Back to top](#)

Flex is a primitive layout component that adaptively aligns child content horizontally or vertically. Flex powers components like `HStack` and `VStack`.

Usage

Flex is used with any of its two sub-components, `FlexItem` and `FlexBlock`.

```
import { Flex, FlexBlock, FlexItem } from '@wordpress/components';

function Example() {
    return (
        <Flex>
            <FlexItem>
                <p>Code</p>
            </FlexItem>
        </Flex>
    );
}
```

```
        </FlexItem>
        <FlexBlock>
            <p>Poetry</p>
        </FlexBlock>
    </Flex>
)
}
```

Props

align: CSSProperties['alignItems']

Aligns children using CSS Flexbox `align-items`. Vertically aligns content if the `direction` is `row`, or horizontally aligns content if the `direction` is `column`.

- Required: No
- Default: `center`

direction: ResponsiveCSSValue<CSSProperties['flexDirection']>

The direction flow of the children content can be adjusted with `direction`. `column` will align children vertically and `row` will align children horizontally.

- Required: No
- Default: `row`

expanded: boolean

Expands to the maximum available width (if horizontal) or height (if vertical).

- Required: No
- Default: `true`

gap: number

Spacing in between each child can be adjusted by using `gap`. The value of `gap` works as a multiplier to the library's grid system (base of 4px).

- Required: No
- Default: 2

justify: CSSProperties['justifyContent']

Horizontally aligns content if the `direction` is `row`, or vertically aligns content if the `direction` is `column`.

- Required: No
- Default: `space-between`

`wrap: boolean`

Determines if children should wrap.

- Required: No
- Default: `false`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Flex”](#)

[Previous FlexItem](#) [Previous: FlexItem](#)
[Next FocalPointPicker](#) [Next: FocalPointPicker](#)

FocalPointPicker

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [url](#)
 - [autoPlay](#)
 - [value](#)
 - [onChange](#)
 - [onDrag](#)
 - [onDragEnd](#)
 - [onDragStart](#)
 - [resolvePoint](#)

[↑ Back to top](#)

Focal Point Picker is a component which creates a UI for identifying the most important visual point of an image. This component addresses a specific problem: with large background images it is common to see undesirable crops, especially when viewing on smaller viewports such as mobile phones. This component allows the selection of the point with the most important visual information and returns it as a pair of numbers between 0 and 1. This value can be easily converted into the CSS `background-position` attribute, and will ensure that the focal point is never cropped out, regardless of viewport.

- Example focal point picker value: `{ x: 0.5, y: 0.1 }`
- Corresponding CSS: `background-position: 50% 10%;`

Usage

```
import { useState } from 'react';
import { FocalPointPicker } from '@wordpress/components';

const Example = () => {
    const [ focalPoint, setFocalPoint ] = useState( {
        x: 0.5,
        y: 0.5,
    } );
    const url = '/path/to/image';

    /* Example function to render the CSS styles based on Focal Point Pick
    const style = {
        backgroundImage: `url(${ url })`,
        backgroundPosition: `${ focalPoint.x * 100 }% ${ focalPoint.y * 100 }%`,
    };

    return (
        <>
            <FocalPointPicker
                url={ url }
                value={ focalPoint }
                onDragStart={ setFocalPoint }
                onDrag={ setFocalPoint }
                onChange={ setFocalPoint }
            />
            <div style={ style } />
        </>
    );
};
```

Props

url

- Type: Text
- Required: Yes

URL of the image or video to be displayed

autoPlay

- Type: Boolean
- Required: No
- Default: true

Autoplays HTML5 video. This only applies to video sources (url).

[value](#)

- Type: Object
- Required: Yes

The focal point. Should be an object containing x and y params.

[onChange](#)

- Type: Function
- Required: Yes

Callback which is called when the focal point changes.

[onDrag](#)

- Type: Function
- Required: No

Callback which is called repetitively during drag operations.

[onDragEnd](#)

- Type: Function
- Required: No

Callback which is called at the end of drag operations.

[onDragStart](#)

- Type: Function
- Required: No

Callback which is called at the start of drag operations.

[resolvePoint](#)

- Type: Function
- Required: No

Function which is called before internal updates to the value state. It receives the upcoming value and may return a modified one.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FocalPointPicker”](#)

FocusableIframe

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [onFocus](#)
 - [iframeRef](#)

[↑ Back to top](#)

Deprecated

`<FocusableIframe />` is a component rendering an `iframe` element enhanced to support focus events. By default, it is not possible to detect when an iframe is focused or clicked within. This enhanced component uses a technique which checks whether the target of a window `blur` event is the iframe, inferring that this has resulted in the focus of the iframe. It dispatches an emulated [FocusEvent](#) on the iframe element with event bubbling, so a parent component binding its own `onFocus` event will account for focus transitioning within the iframe.

Usage

Use as you would a standard `iframe`. You may pass `onFocus` directly as the callback to be invoked when the iframe receives focus, or on an ancestor component since the event will bubble.

```
import { FocusableIframe } from '@wordpress/components';

const MyFocusableIframe = () => (
  <FocusableIframe
    src="/my-iframe-url"
    onFocus={ () => console.log( 'iframe is focused' ) }
  />
);
```

Props

Any props aside from those listed below will be passed to the `FocusableIframe` will be passed through to the underlying `iframe` element.

[onFocus](#)

- Type: Function
- Required: No

Callback to invoke when iframe receives focus. Passes an emulated FocusEvent object as the first argument.

[iframeRef](#)

- Type: `React.Ref`
- Required: No

If a reference to the underlying DOM element is needed, pass `iframeRef` as the result of a `React.createRef` called from your component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FocusableIframe”](#)

[Previous FocalPointPicker](#) [Previous: FocalPointPicker](#)
[Next FontSizePicker](#) [Next: FontSizePicker](#)

FontSizePicker

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [disableCustomFontSizes: boolean](#)
 - [fallbackFontSize: number](#)
 - [fontSizes: FontSize\[\]](#)
 - [onChange: \(value: number | string | undefined, selectedItem?: FontSize \) => void](#)
 - [size: ‘default’ | ‘unstable-large’](#)
 - [units: string\[\]](#)
 - [value: number | string](#)
 - [withReset: boolean](#)
 - [withSlider: boolean](#)
 - [_nextHasNoMarginBottom: boolean](#)

[↑ Back to top](#)

FontSizePicker is a React component that renders a UI that allows users to select a font size. The component renders a user interface that allows the user to select predefined (common) font sizes and contains an option that allows users to select custom font sizes (by choosing the value) if that functionality is enabled.

Usage

```
import { useState } from 'react';
import { FontSizePicker } from '@wordpress/components';
import { __ } from '@wordpress/i18n';

const fontSizes = [
  {
    name: __( 'Small' ),
    slug: 'small',
    size: 12,
  },
  {
    name: __( 'Big' ),
    slug: 'big',
    size: 26,
  },
];
const fallbackFontSize = 16;

const MyFontSizePicker = () => {
  const [ fontSize, setFontSize ] = useState( 12 );

  return (
    <FontSizePicker
      __nextHasNoMarginBottom
      fontSizes={ fontSizes }
      value={ fontSize }
      fallbackFontSize={ fallbackFontSize }
      onChange={ ( newFontSize ) => {
        setFontSize( newFontSize );
      } }
    />
  );
};

...
<MyFontSizePicker />
```

Props

The component accepts the following props:

[disableCustomFontSizes: boolean](#)

If `true`, it will not be possible to choose a custom `fontSize`. The user will be forced to pick one of the pre-defined sizes passed in `fontSizes`.

- Required: no
- Default: `false`

[fallbackFontSize: number](#)

If no value exists, this prop defines the starting position for the font size picker slider. Only relevant if `withSlider` is `true`.

- Required: No

[fontSizes: FontSize\[\]](#)

An array of font size objects. The object should contain properties `size`, `name`, and `slug`. The property `size` contains a number with the font size value, in px or a string specifying the font size CSS property that should be used eg: “13px”, “1em”, or “clamp(12px, 5vw, 100px)”. The `name` property includes a label for that font size e.g.: `Small`. The `slug` property is a string with a unique identifier for the font size. Used for the class generation process.

Note: The slugs `default` and `custom` are reserved and cannot be used.

- Required: No
- Default: []

[onChange: \(value: number | string | undefined, selectedItem?: FontSize \) => void](#)

A function that receives the new font size value.

If `onChange` is called without any parameter, it should reset the value, attending to what reset means in that context, e.g., set the font size to `undefined` or set the font size a starting value.

- Required: Yes

[size: ‘default’ | ‘unstable-large’](#)

Size of the control.

- Required: No
- Default: 'default'

[units: string\[\]](#)

Available units for custom font size selection.

- Required: No

[value: number | string](#)

The current font size value.

- Required: No

[withReset: boolean](#)

If `true`, a reset button will be displayed alongside the input field when a custom font size is active. Has no effect when `disableCustomFontSizes` is `true`.

- Required: no
- Default: `true`

[withSlider: boolean](#)

If `true`, a slider will be displayed alongside the input field when a custom font size is active. Has no effect when `disableCustomFontSizes` is `true`.

- Required: no
- Default: `false`

[nextHasNoMarginBottom: boolean](#)

Start opting into the new margin-free styles that will become the default in a future version, currently scheduled to be WordPress 6.4. (The prop can be safely removed once this happens.)

- Required: no
- Default: `false`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FontSizePicker”](#)

[Previous FocusableIframe](#) [Previous: FocusableIframe](#)
[Next FormFileUpload](#) [Next: FormFileUpload](#)

FormFileUpload

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [accept](#)
 - [children](#)
 - [icon](#)
 - [multiple](#)
 - [onChange](#)

- [onClick](#)
- [render](#)

[↑ Back to top](#)

FormFileUpload is a component that allows users to select files from their local device.

Usage

```
import { FormFileUpload } from '@wordpress/components';

const MyFormFileUpload = () => (
  <FormFileUpload
    accept="image/*"
    onChange={ ( event ) => console.log( event.currentTarget.files ) }
  >
    Upload
  </FormFileUpload>
);


```

Props

The component accepts the following props. Props not included in this set will be passed to the `Button` component.

accept

A string passed to `input` element that tells the browser which file types can be upload to the upload by the user use. e.g: `image/*,video/*`.

More information about this string is available in https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/file#Unique_file_type_specifiers.

- Type: `String`
- Required: No

children

Children are passed as children of `Button`.

- Type: `Boolean`
- Required: No

icon

The icon to render. Supported values are: Dashicons (specified as strings), functions, Component instances and `null`.

- Type: `String|Function|Component|null`
- Required: No
- Default: `null`

[multiple](#)

Whether to allow multiple selection of files or not.

- Type: Boolean
- Required: No
- Default: false

[onChange](#)

Callback function passed directly to the `input` file element.

Select files will be available in `event.currentTarget.files`.

- Type: Function
- Required: Yes

[onClick](#)

Callback function passed directly to the `input` file element.

This can be useful when you want to force a `change` event to fire when the user chooses the same file again. To do this, set the target value to an empty string in the `onClick` function.

```
<FormFileUpload
    onClick={ ( event ) => ( event.target.value = '' ) }
    onChange={ onChange }
>
    Upload
</FormFileUpload>
```

- Type: Function
- Required: No

[render](#)

Optional callback function used to render the UI. If passed, the component does not render the default UI (a button) and calls this function to render it. The function receives an object with property `openFileDialog`, a function that, when called, opens the browser native file upload modal window.

- Type: Function
- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FormFileUpload”](#)

[Previous FontSizePicker](#) [Previous: FontSizePicker](#)

[Next FormToggle](#) [Next: FormToggle](#)

FormToggle

In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
 - [Behavior](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

FormToggle switches a single setting on or off.



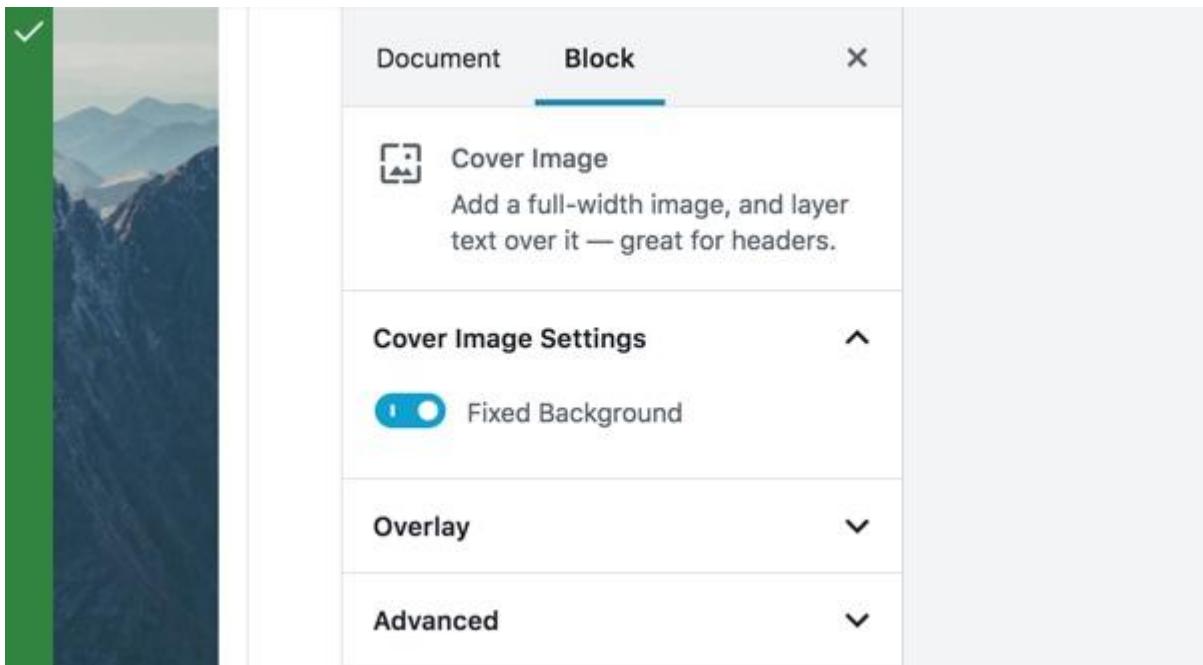
Design guidelines

Usage

When to use toggles

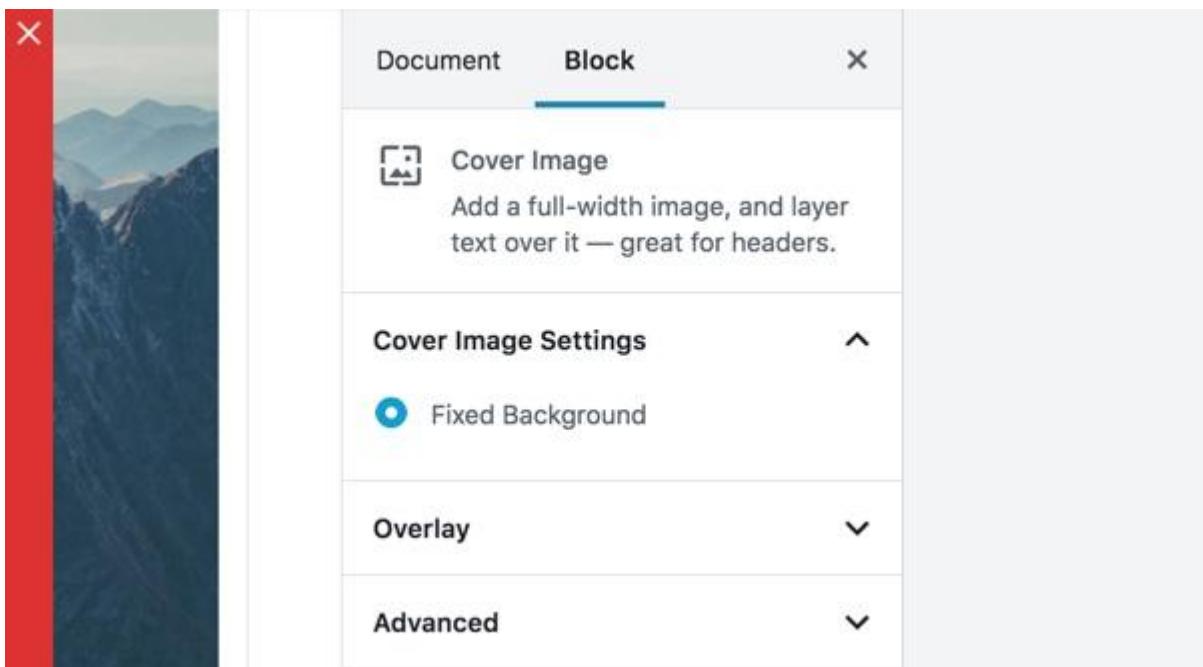
Use toggles when you want users to:

- Switch a single option on or off.
- Immediately activate or deactivate something.



Do

Use toggles to switch an option on or off.



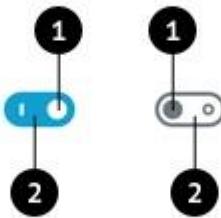
Don't

Don't use radio buttons for settings that toggle on and off.

Toggles are preferred when the user is not expecting to submit data, as is the case with checkboxes and radio buttons.

State

When the user slides a toggle thumb (1) to the other side of the track (2) and the state of the toggle changes, it's been successfully toggled.



Text label

Toggles should have clear inline labels so users know exactly what option the toggle controls, and whether the option is enabled or disabled.

Do not include any text (e.g. “on” or “off”) within the toggle element itself. The toggle alone should be sufficient to communicate the state.

Behavior

When a user switches a toggle, its corresponding action takes effect immediately.

Development guidelines

Usage

```
import { useState } from 'react';
import { FormToggle } from '@wordpress/components';

const MyFormToggle = () => {
    const [ isChecked, setChecked ] = useState( true );

    return (
        <FormToggle
            checked={ isChecked }
            onChange={ () => setChecked( ( state ) => ! state ) }
        />
    );
};
```

Props

The component accepts the following props:

`checked: boolean`

If checked is true the toggle will be checked. If checked is false the toggle will be unchecked. If no value is passed the toggle will be unchecked.

- Required: No

`disabled: boolean`

If disabled is true the toggle will be disabled and apply the appropriate styles.

- Required: No

`onChange: (event: ChangeEvent<HTMLInputElement>) => void`

A callback function invoked when the toggle is clicked.

- Required: Yes

Related components

- To select one option from a set, and you want to show them all the available options at once, use the `Radio` component.
- To select one or more items from a set, use the `CheckboxControl` component.
- To display a toggle with label and help text, use the `ToggleControl` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FormToggle](#)

[Previous FormFileUpload](#) [Previous: FormFileUpload](#)

[Next FormTokenField](#) [Next: FormTokenField](#)

FormTokenField

In this article

Table of Contents

- [Keyboard Accessibility](#)
- [Properties](#)
- [Usage](#)

[↑ Back to top](#)

A `FormTokenField` is a field similar to the tags and categories fields in the interim editor chrome, or the “to” field in Mail on OS X. Tokens can be entered by typing them or selecting them from a list of suggested tokens.

Up to one hundred suggestions that match what the user has typed so far will be shown from which the user can pick from (auto-complete). Tokens are separated by the “,” character. Suggestions can be selected with the up or down arrows and added with the tab or enter key.

The `value` property is handled in a manner similar to controlled form components. See [Forms](#) in the React Documentation for more information.

Keyboard Accessibility

- `left arrow` – if input field is empty, move insertion point before previous token
- `right arrow` – if input field is empty, move insertion point after next token
- `up arrow` – select previous suggestion
- `down arrow` – select next suggestion
- `tab / enter` – if suggestion selected, insert suggestion as a new token; otherwise, insert value typed into input as new token
- `comma` – insert value typed into input as new token

Properties

- `value` – An array of strings or objects to display as tokens in the field. If objects are present in the array, they **must** have a property of `value`. Here is an example object that could be passed in as a value:

```
{
```

```
  value: '(string) The value of the token.',  
  status: "(string) One of 'error', 'validating', or 'success'. Applies  
  title: '(string) If not falsey, will add a title to the token.',  
  onMouseEnter: '(function) Function to call when onMouseEnter event tr  
  onMouseLeave: '(function) Function to call when onMouseLeave is triggered
```

```
}
```

- `displayTransform` – Function to call to transform tokens for display. (In the editor, this is needed to decode HTML entities embedded in tags – otherwise entities like & in tag names are double-encoded like & ;, once by the REST API and once by React).
- `saveTransform` – Function to call to transform tokens for saving. The default is to trim the token value. This function is also applied when matching suggestions against the current value so that matching works correctly with leading or trailing spaces. (In the editor, this is needed to remove leading and trailing spaces from tag names, like wp-admin does. Otherwise the REST API won’t save them.)
- `onChange` – Function to call when the tokens have changed. An array of new tokens is passed to the callback.
- `onInputChange` – Function to call when the users types in the input field. It can be used to trigger autocomplete requests.

- `onFocus` – Function to call when the TokenField has been focused on. The event is passed to the callback. Useful for analytics.
- `suggestions` – An array of strings to present to the user as suggested tokens.
- `maxSuggestions` – The maximum number of suggestions to display at a time.
- `tokenizeOnSpace` – If true, will add a token when TokenField is focused and space is pressed.
- `isBorderless` – When true, renders tokens as without a background.
- `maxLength` – If passed, TokenField will disable ability to add new tokens once number of tokens is greater than or equal to `maxLength`.
- `disabled` – When true, tokens are not able to be added or removed.
- `placeholder` – If passed, the TokenField input will show a placeholder string if no value tokens are present.
- `messages` – Allows customizing the messages presented by screen readers in different occasions:
 - `added` – The user added a new token.
 - `removed` – The user removed an existing token.
 - `remove` – The user focused the button to remove the token.
 - `experimentalInvalid` – The user tried to add a token that didn't pass the validation.
- `experimentalRenderItem` – Custom renderer invoked for each option in the suggestion list. The render prop receives as its argument an object containing, under the `item` key, the single option's data (directly from the array of data passed to the `options` prop).
- `experimentalExpandOnFocus` – If true, the suggestions list will be always expanded when the input field has the focus.
- `experimentalShowHowTo` – If false, the text on how to use the select (ie: *Separate with commas or the Enter key.*) will be hidden.
- `experimentalValidateInput` – If passed, all introduced values will be validated before being added as tokens.
- `experimentalAutoSelectFirstMatch` – If true, the select the first matching suggestion when the user presses the Enter key (or space when tokenizeOnSpace is true).
- `nextHasNoMarginBottom` – Start opting into the new margin-free styles that will become the default in a future version, currently scheduled to be WordPress 6.5. (The prop can be safely removed once this happens.)
- `tokenizeOnBlur` – If true, add any incompleteTokenValue as a new token when the field loses focus.

Usage

```

import { useState } from 'react';
import { FormTokenField } from '@wordpress/components';

const continents = [
  'Africa',
  'America',
  'Antarctica',
  'Asia',
  'Europe',
  'Oceania',
];
const MyFormTokenField = () => {

```

```
const [ selectedContinents, setSelectedContinents ] = useState( [] );

return (
  <FormTokenField
    value={ selectedContinents }
    suggestions={ continents }
    onChange={ ( tokens ) => setSelectedContinents( tokens ) }
  />
);
};
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: FormTokenField](#)

[Previous FormToggle](#) [Previous: FormToggle](#)
[Next GradientPicker](#) [Next: GradientPicker](#)

GradientPicker

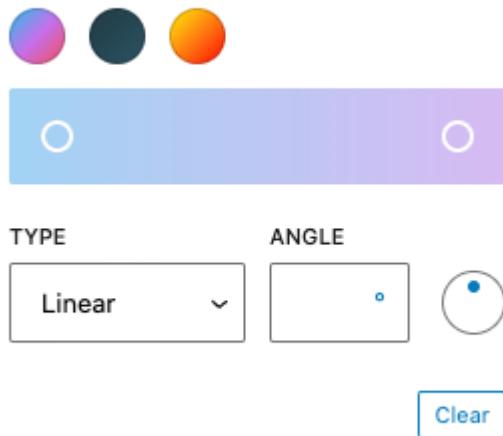
In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [className: string](#)
 - [value: string](#)
 - [onChange: \(currentGradient: string | undefined \) => void](#)
 - [gradients: GradientsProp\[\]](#)
 - [clearable: boolean](#)
 - [disableCustomGradients: boolean](#)
 - [nextHasNoMargin: boolean](#)
 - [headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | '1' | '2' | '3' | '4' | '5' | '6'](#)
 - [asButtons: boolean](#)
 - [loop: boolean](#)

[↑ Back to top](#)

GradientPicker is a React component that renders a color gradient picker to define a multi step gradient. There's either a *linear* or a *radial* type available.



Usage

Render a GradientPicker.

```
import { useState } from 'react';
import { GradientPicker } from '@wordpress/components';

const myGradientPicker = () => {
    const [ gradient, setGradient ] = useState( null );

    return (
        <GradientPicker
            __nextHasNoMargin
            value={ gradient }
            onChange={ ( currentGradient ) => setGradient( currentGradient )
            gradients={ [
                {
                    name: 'JShine',
                    gradient:
                        'linear-gradient(135deg,#12c2e9 0%,#c471ed 50%,#f6
                        slug: 'jshine',
                },
                {
                    name: 'Moonlit Asteroid',
                    gradient:
                        'linear-gradient(135deg,#0F2027 0%, #203A43 0%, #2
                        slug: 'moonlit-asteroid',
                },
                {
                    name: 'Rastafarie',
                    gradient:
                        'linear-gradient(135deg,#1E9600 0%, #FFF200 0%, #F
                        slug: 'rastafari',
                },
            ],
        }
    );
}
```

```
        ] }
      />
    );
};
```

Props

The component accepts the following props:

className: string

The class name added to the wrapper.

- Required: No

value: string

The current value of the gradient. Pass a css gradient like `linear-gradient(90deg, rgb(6, 147, 227) 0%, rgb(155, 81, 224) 100%)`. Optionally pass in a null value to specify no gradient is currently selected.

- Required: No
- Default: `linear-gradient(90deg, rgb(6, 147, 227) 0%, rgb(155, 81, 224) 100%)`

onChange: (currentGradient: string | undefined) => void

The function called when a new gradient has been defined. It is passed the `currentGradient` as an argument.

- Required: Yes

gradients: GradientsProp[]

An array of objects of predefined gradients displayed above the gradient selector.

- Required: No
- Default: []

clearable: boolean

Whether the palette should have a clearing button or not.

- Required: No
- Default: true

disableCustomGradients: boolean

If true, the gradient picker will not be displayed and only defined gradients from `gradients` are available.

- Required: No

- Default: false

[nextHasNoMargin: boolean](#)

Start opting into the new margin-free styles that will become the default in a future version, currently scheduled to be WordPress 6.4. (The prop can be safely removed once this happens.)

- Required: No
- Default: `false`

[headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | '1' | '2' | '3' | '4' | '5' | '6'](#)

The heading level. Only applies in cases where gradients are provided from multiple origins (ie. when the array passed as the `gradients` prop contains two or more items).

- Required: No
- Default: 2

[asButtons: boolean](#)

Whether the control should present as a set of buttons, each with its own tab stop.

- Required: No
- Default: `false`

[loop: boolean](#)

Prevents keyboard interaction from wrapping around. Only used when `asButtons` is not true.

- Required: No
- Default: `true`

First published

December 28, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: GradientPicker”](#)

[Previous FormTokenField](#) [Previous: FormTokenField](#)

[Next Grid](#) [Next: Grid](#)

Grid

In this article

Table of Contents

- [Usage](#)
- [Props](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Grid is a primitive layout component that can arrange content in a grid configuration.

[Usage](#)

```
import {
    __experimentalGrid as Grid,
    __experimentalText as Text,
} from '@wordpress/components';

function Example() {
    return (
        <Grid columns={ 3 }>
            <Text>Code</Text>
            <Text>is</Text>
            <Text>Poetry</Text>
        </Grid>
    );
}
```

[Props](#)

`align: CSS['alignItems']`

Adjusts the block alignment of children.

- Required: No

`alignment: GridAlignment`

Adjusts the horizontal and vertical alignment of children.

- Required: No

```
columnGap: CSSProperties['gridColumnGap']
```

Adjusts the `grid-column-gap`.

- Required: No

```
columns: number
```

Adjusts the number of columns of the Grid.

- Required: No
- Default: 2

```
gap: number
```

Gap between each child.

- Required: No
- Default: 3

```
isInline: boolean
```

Changes the CSS display from `grid` to `inline-grid`.

- Required: No

```
justify: CSS['justifyContent']
```

Adjusts the inline alignment of children.

- Required: No

```
rowGap: CSSProperties['gridRowGap']
```

Adjusts the `grid-row-gap`.

- Required: No

```
rows: number
```

Adjusts the number of rows of the Grid.

- Required: No

```
templateColumns: CSS['gridTemplateColumns']
```

Adjusts the CSS grid `template-columns`.

- Required: No

```
templateRows: CSS['gridTemplateRows']
```

Adjusts the CSS grid `template-rows`.

- Required: No

First published

May 7, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Grid”](#)

[Previous GradientPicker](#) [Previous: GradientPicker](#)

[Next Guide](#) [Next: Guide](#)

Guide

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [className](#)
 - [contentLabel](#)
 - [finishButtonText](#)
 - [onFinish](#)
 - [pages](#)

[↑ Back to top](#)

Guide is a React component that renders a *user guide* in a modal. The guide consists of several pages which the user can step through one by one. The guide is finished when the modal is closed or when the user clicks *Finish* on the last page of the guide.

[Usage](#)

```
function MyTutorial() {  
  const [ isOpen, setIsOpen ] = useState( true );  
  
  if ( ! isOpen ) {  
    return null;  
  }  
  
  return (  
    <div>
```

```

<Guide
    onFinish={ () => setIsOpen( false ) }
    pages={ [
        {
            content: <p>Welcome to the ACME Store!</p>,
        },
        {
            image: 
            content: (
                <p>
                    Click <i>Add to Cart</i> to buy a product.
                </p>
            ),
        },
    ] }
/>
);
}

```

Props

The component accepts the following props:

className

A custom class to add to the modal.

- Type: `string`
- Required: No

contentLabel

This property is used as the modal's accessibility label. It is required for accessibility reasons.

- Type: `string`
- Required: Yes

finishButtonText

Use this to customize the label of the *Finish* button shown at the end of the guide.

- Type: `string`
- Required: No
- Default: 'Finish'

onFinish

A function which is called when the guide is finished. The guide is finished when the modal is closed or when the user clicks *Finish* on the last page of the guide.

- Type: `(event?: KeyboardEvent< HTMLDivElement > | SyntheticEvent) => void`
- Required: Yes

pages

A list of objects describing each page in the guide. Each object **must** contain a 'content' property and may optionally contain a 'image' property.

- Type: { content: ReactNode; image?: ReactNode; }[]
- Required: No
- Default: []

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Guide”](#)

[Previous Grid](#) [Previous: Grid](#)
[Next HStack](#) [Next: HStack](#)

HStack

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
- [Spacer](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

HStack (Horizontal Stack) arranges child elements in a horizontal line.

Usage

HStack can render anything inside.

```
import {
  __experimentalHStack as HStack,
  __experimentalText as Text,
} from '@wordpress/components';

function Example() {
```

```
    return (
      <HStack>
        <Text>Code</Text>
        <Text>is</Text>
        <Text>Poetry</Text>
      </HStack>
    );
}
```

Props

alignment

Type: HStackAlignment | CSS['alignItems']

Determines how the child elements are aligned.

- **top**: Aligns content to the top.
- **topLeft**: Aligns content to the top/left.
- **topRight**: Aligns content to the top/right.
- **left**: Aligns content to the left.
- **center**: Aligns content to the center.
- **right**: Aligns content to the right.
- **bottom**: Aligns content to the bottom.
- **bottomLeft**: Aligns content to the bottom/left.
- **bottomRight**: Aligns content to the bottom/right.
- **edge**: Justifies content to be evenly spread out up to the main axis edges of the container.
- **stretch**: Stretches content to the cross axis edges of the container.

direction

Type: FlexDirection

The direction flow of the children content can be adjusted with `direction`. `column` will align children vertically and `row` will align children horizontally.

expanded

Type: boolean

Expands to the maximum available width (if horizontal) or height (if vertical).

justify

Type: CSS['justifyContent']

Horizontally aligns content if the `direction` is `row`, or vertically aligns content if the `direction` is `column`.

In the example below, `flex-start` will align the children content to the left.

spacing

Type: CSS['width']

The amount of space between each child element. Spacing in between each child can be adjusted by using `spacing`.

The value of `spacing` works as a multiplier to the library's grid system (base of 4px).

wrap

Type: boolean

Determines if children should wrap.

Spacer

When a `Spacer` is used within an `HStack`, the `Spacer` adaptively expands to take up the remaining space.

```
import {
  __experimentalHStack as HStack,
  __experimentalSpacer as Spacer,
  __experimentalText as Text,
} from '@wordpress/components';

function Example() {
  return (
    <HStack>
      <Text>Code</Text>
      <Spacer>
        <Text>is</Text>
      </Spacer>
      <Text>Poetry</Text>
    </HStack>
  );
}
```

`Spacer` also be used in-between items to push them apart.

```
import {
  __experimentalHStack as HStack,
  __experimentalSpacer as Spacer,
  __experimentalText as Text,
} from '@wordpress/components';

function Example() {
  return (
    <HStack>
      <Text>Code</Text>
      <Spacer />
      <Text>is</Text>
      <Text>Poetry</Text>
    </HStack>
  );
}
```

```
) ;  
}
```

First published

May 6, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: HStack](#)

[Previous Guide](#) [Previous: Guide](#)
[Next Heading](#) [Next: Heading](#)

Heading

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`Heading` renders headings and titles using the library’s typography system.

[Usage](#)

```
import { __experimentalHeading as Heading } from '@wordpress/components';

function Example() {
    return <Heading>Code is Poetry</Heading>;
}
```

[Props](#)

`Heading` uses `Text` underneath, so we have access to all of `Text`’s props except for:

- `size` which is replaced by `level`;
- `isBlock`’s default value, which is `true` for the `Heading` component.

For a complete list of those props, check out [Text](#).

```
level: 1 | 2 | 3 | 4 | 5 | 6 | '1' | '2' | '3' | '4' | '5' | '6'
```

Passing any of the heading levels to `level` will both render the correct typographic text size as well as the semantic element corresponding to the level (`h1` for 1 for example).

- Required: No
- Default: 2

First published

April 30, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Heading”](#)

[Previous HStack](#) [Previous: HStack](#)

[Next NavigateRegions](#) [Next: NavigateRegions](#)

NavigateRegions

In this article

[Table of Contents](#)

- [Example:](#)
- [Notes:](#)

[↑ Back to top](#)

`navigateRegions` is a React [higher-order component](#) adding keyboard navigation to switch between the different DOM elements marked as “regions” (`role=”region”`). These regions should be focusable (By adding a `tabIndex` attribute for example). For better accessibility, these elements must be properly labelled to briefly describe the purpose of the content in the region. For more details, see “Landmark Roles” in the [WAI-ARIA specification](#) and “Landmark Regions” in the [ARIA Authoring Practices Guide](#).

[Example:](#)

```
import { navigateRegions } from '@wordpress/components';

const MyComponentWithNavigateRegions = navigateRegions( () => (
  <div>
    <div role="region" tabIndex="-1" aria-label="Header">
      Header
    </div>
    <div role="region" tabIndex="-1" aria-label="Content">
```

```
        Content
    </div>
    <div role="region" tabIndex="-1" aria-label="Sidebar">
        Sidebar
    </div>
</div>
) );
```

Notes:

It's important to note that an ARIA `role="region"` is an ARIA landmark role. It should be reserved for sections of content sufficiently important to have it listed in a summary of the page. Only use this ARIA role for the main sections of a page. All perceivable content should reside in a semantically meaningful landmark in order that content is not missed by the user.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NavigateRegions”](#)

[Previous Heading](#) [Previous: Heading](#)
[Next HigherOrder](#) [Next: HigherOrder](#)

HigherOrder

[↑ Back to top](#)

This directory includes a library of generic Higher Order React Components.

[Learn more about Higher Order Components](#)

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: HigherOrder”](#)

[Previous NavigateRegions](#) [Previous: NavigateRegions](#)

WithConstrainedTabbing

[↑ Back to top](#)

`withConstrainedTabbing` is a React [higher-order component](#) adding the ability to constrain keyboard navigation with the Tab key within a component. For accessibility reasons, some UI components need to constrain Tab navigation, for example modal dialogs or similar UI. Use of this component is recommended only in cases where a way to navigate away from the wrapped component is implemented by other means, usually by pressing the Escape key or using a specific UI control, e.g. a “Close” button.

Usage

Wrap your original component with `withConstrainedTabbing`.

```
import { useState } from 'react';
import {
  withConstrainedTabbing,
  TextControl,
  Button,
} from '@wordpress/components';

const ConstrainedTabbing = withConstrainedTabbing(
  ( { children } ) => children
);

const MyComponentWithConstrainedTabbing = () => {
  const [ isConstrainedTabbing, setIsConstrainedTabbing ] = useState( false );
  let form = (
    <form>
      <TextControl label="Input 1" onChange={ () => {} } />
      <TextControl label="Input 2" onChange={ () => {} } />
    </form>
  );
  if ( isConstrainedTabbing ) {
    form = <ConstrainedTabbing>{ form }</ConstrainedTabbing>;
  }

  const toggleConstrain = () => {
    setIsConstrainedTabbing( ( state ) => ! state );
  };

  return (
    <div>
      { form }
      <Button variant="secondary" onClick={ toggleConstrain }>
        { isConstrainedTabbing ? 'Disable' : 'Enable' } constrain
      </Button>
    </div>
  );
}
```

```
        tabbing
    </Button>
</div>
);
}
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WithConstrainedTabbing”](#)

[Previous HigherOrder](#) [Previous: HigherOrder](#)
[Next WithFallbackStyles](#) [Next: WithFallbackStyles](#)

WithFallbackStyles

[↑ Back to top](#)

Usage

```
import { withFallbackStyles, Button } from '@wordpress/components';

const { getComputedStyle } = window;

const MyComponentWithFallbackStyles = withFallbackStyles(
( node, ownProps ) => {
    const buttonNode = node.querySelector( 'button' );
    return {
        fallbackBackgroundColor: getComputedStyle( buttonNode )
            .backgroundColor,
        fallbackTextColor: getComputedStyle( buttonNode ).color,
    };
}
)( ( { fallbackTextColor, fallbackBackgroundColor } ) => (
<div>
    <Button variant="primary">My button</Button>
    <div>Text color: { fallbackTextColor }</div>
    <div>Background color: { fallbackBackgroundColor }</div>
</div>
) );
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WithFallbackStyles”](#)

[Previous WithConstrainedTabbing](#) [Previous: WithConstrainedTabbing](#)

[Next WithFilters](#) [Next: WithFilters](#)

WithFilters

[↑ Back to top](#)

`withFilters` is a part of [Native Gutenberg Extensibility](#). It is also a React [higher-order component](#).

Wrapping a component with `withFilters` provides a filtering capability controlled externally by the `hookName`.

Usage

```
import { withFilters } from '@wordpress/components';
import { addFilter } from '@wordpress/hooks';

const MyComponent = ( { title } ) => <h1>{ title }</h1>;

const ComponentToAppend = () => <div>Appended component</div>;

function withComponentAppended( FilteredComponent ) {
    return ( props ) => (
        <>
            <FilteredComponent { ...props } />
            <ComponentToAppend />
        </>
    );
}

addFilter(
    'MyHookName',
    'my-plugin/with-component-appended',
    withComponentAppended
);

const MyComponentWithFilters = withFilters( 'MyHookName' )( MyComponent );
```

`withFilters` expects a string argument which provides a hook name. It returns a function which can then be used in composing your component. The hook name allows plugin developers to customize or completely override the component passed to this higher-order component using `wp.hooks.addFilter` method.

It is also possible to override props by implementing a higher-order component which works as follows:

```
import { withFilters } from '@wordpress/components';
import { addFilter } from '@wordpress/hooks';

const MyComponent = ( { hint, title } ) => (
    <>
        <h1>{ title }</h1>
        <p>{ hint }</p>
    </>
);

function withHintOverridden( FilteredComponent ) {
    return ( props ) => (
        <FilteredComponent { ...props } hint="Overridden hint" />
    );
}

addFilter( 'MyHookName', 'my-plugin/with-hint-overridden', withHintOverriden );

const MyComponentWithFilters = withFilters( 'MyHookName' )( MyComponent );
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WithFilters](#)

[Previous WithFallbackStyles](#) [Previous: WithFallbackStyles](#)

[Next WithFocusOutside](#) [Next: WithFocusOutside](#)

WithFocusOutside

[↑ Back to top](#)

`WithFocusOutside` is a React [higher-order component](#) to enable behavior to occur when focus leaves an element. Since a `blur` event will fire in React even when focus transitions to another element in the same context, this higher-order component encapsulates the logic necessary to determine if focus has truly left the element.

Usage

Wrap your original component with `withFocusOutside`, defining a `handleFocusOutside` instance method on the component class.

Note: `withFocusOutside` must only be used to wrap the Component class.

```
import { withFocusOutside, TextControl } from '@wordpress/components';

const MyComponentWithFocusOutside = withFocusOutside(
  class extends React.Component {
    handleFocusOutside() {
      console.log( 'Focus outside' );
    }

    render() {
      return (
        <div>
          <TextControl onChange={ () => {} } />
          <TextControl onChange={ () => {} } />
        </div>
      );
    }
  );
);
```

In the above example, the `handleFocusOutside` function is only called if focus leaves the element, and not if transitioning focus between the two inputs.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WithFocusOutside](#)

[Previous WithFilters](#) [Previous: WithFilters](#)
[Next WithFocusReturn](#) [Next: WithFocusReturn](#)

WithFocusReturn

In this article

Table of Contents

- [Usage](#)
 - [withFocusReturn](#)

[↑ Back to top](#)

`withFocusReturn` is a higher-order component used typically in scenarios of short-lived elements (modals, dropdowns) where, upon the element's unmounting, focus should be restored to the focused element which had initiated it being rendered.

Usage

withFocusReturn

```
import { useState } from 'react';
import { withFocusReturn, TextControl, Button } from '@wordpress/component'

const EnhancedComponent = withFocusReturn( () => (
  <div>
    Focus will return to the previous input when this component is unmounted
    <TextControl autoFocus={ true } onChange={ () => {} } />
  </div>
) );

const MyComponentWithFocusReturn = () => {
  const [ text, setText ] = useState( '' );
  const unmount = () => {
    document.activeElement.blur();
    setText( '' );
  };

  return (
    <div>
      <TextControl
        placeholder="Type something"
        value={ text }
        onChange={ ( value ) => setText( value ) }
      />
      { text && <EnhancedComponent /> }
      { text && (
        <Button variant="secondary" onClick={ unmount }>
          Unmount
        </Button>
      ) }
    </div>
  );
}
```

`withFocusReturn` can optionally be called as a higher-order function creator. Provided an options object, a new higher-order function is returned.

Currently, the following options are supported:

onFocusReturn

An optional function which allows the developer to customize the focus return behavior. A return value of `false` should be returned from this function to indicate that the default focus return behavior should be skipped.

- Type: Function
- Required: No

Example:

```
function MyComponent() {  
    return <textarea />;  
}  
  
const EnhancedMyComponent = withFocusReturn( {  
    onFocusReturn() {  
        document.getElementById( 'other-input' ).focus();  
        return false;  
    },  
} )( MyComponent );
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WithFocusReturn”](#)

[Previous WithFocusOutside](#) [Previous: WithFocusOutside](#)
[Next WithNotices](#) [Next: WithNotices](#)

WithNotices

[↑ Back to top](#)

`withNotices` is a React [higher-order component](#) used typically in adding the ability to post notice messages within the original component.

Wrapping the original component with `withNotices` encapsulates the component with the additional props `noticeOperations`, `noticeUI`, and `noticeList`.

noticeOperations

Contains a number of useful functions to add notices to your site.

createNotice

Function passed down as a prop that adds a new notice.

Parameters

- *notice* object: Notice to add.

createErrorNotice

Function passed as a prop that adds a new error notice.

Parameters

- *msg* string: Error message of the notice.

removeAllNotices

Function that removes all notices.

removeNotice

Function that removes notice by ID.

Parameters

- *id* string: ID of notice to remove.

#noticeUi

The rendered `NoticeList`.

#noticeList

The array of notice objects to be displayed.

Usage

```
import { withNotices, Button } from '@wordpress/components';

const MyComponentWithNotices = withNotices(
  ( { noticeOperations, noticeUI } ) => {
    const addError = () =>
      noticeOperations.createErrorNotice( 'Error message' );
    return (
      <div>
        { noticeUI }
        <Button variant="secondary" onClick={ addError }>
          Add error
        </Button>
      </div>
    );
  }
);
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WithNotices”](#)

[Previous WithFocusReturn](#) [Previous: WithFocusReturn](#)
[Next WithSpokenMessages](#) [Next: WithSpokenMessages](#)

WithSpokenMessages

[↑ Back to top](#)

Usage

```
import { withSpokenMessages, Button } from '@wordpress/components';

const MyComponentWithSpokenMessages = withSpokenMessages(
  ( { speak, debouncedSpeak } ) => (
    <div>
      <Button
        variant="secondary"
        onClick={ () => speak( 'Spoken message' ) }
      >
        Speak
      </Button>
      <Button
        variant="secondary"
        onClick={ () => debouncedSpeak( 'Delayed message' ) }
      >
        Debounced Speak
      </Button>
    </div>
  )
);
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WithSpokenMessages”](#)

[Previous WithNotices](#) [Previous: WithNotices](#)
[Next Icon](#) [Next: Icon](#)

Icon

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [icon](#)
 - [size](#)

[↑ Back to top](#)

Allows you to render a raw icon without any initial styling or wrappers.

[Usage](#)

With a Dashicon

```
import { Icon } from '@wordpress/components';

const MyIcon = () => <Icon icon="screenoptions" />;
```

With a function

```
import { Icon } from '@wordpress/components';

const MyIcon = () => (
  <Icon
    icon={ () => (
      <svg>
        <path d="M5 4v3h5.5v12h3V7H19V4z" />
      </svg>
    ) }
  />
);
```

With a Component

```
import { MyIconComponent } from '../my-icon-component';
import { Icon } from '@wordpress/components';

const MyIcon = () => <Icon icon={ MyIconComponent } />;
```

With an SVG

```
import { Icon } from '@wordpress/components';

const MyIcon = () => (
```

```
<Icon
  icon={
    <svg>
      <path d="M5 4v3h5.5v12h3V7H19V4z" />
    </svg>
  }
/>
);
```

Specifying a className

```
import { Icon } from '@wordpress/components';

const MyIcon = () => <Icon icon="screenoptions" className="example-class"
```

Props

The component accepts the following props. Any additional props are passed through to the underlying icon element.

icon

The icon to render. Supported values are: Dashicons (specified as strings), functions, Component instances and `null`.

- Type: `String|Function|Component|null`
- Required: No
- Default: `null`

size

The size (width and height) of the icon.

- Type: `Number`
- Required: No
- Default: `20` when a Dashicon is rendered, `24` for all other icons.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Icon”](#)

[Previous WithSpokenMessages](#) [Previous: WithSpokenMessages](#)
[Next InputControl](#) [Next: InputControl](#)

InputControl

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [disabled](#)
 - [isPressEnterToChange](#)
 - [hideLabelFromVision](#)
 - [label](#)
 - [labelPosition](#)
 - [onChange](#)
 - [prefix](#)
 - [size](#)
 - [suffix](#)
 - [type](#)
 - [value](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

InputControl components let users enter and edit text. This is an experimental component intended to (in time) merge with or replace [TextControl](#).

[Usage](#)

```
import { __experimentalInputControl as InputControl } from '@wordpress/com
import { useState } from '@wordpress/compose';

const Example = () => {
    const [ value, setValue ] = useState( '' );

    return (
        <InputControl
            value={ value }
            onChange={ ( nextValue ) => setValue( nextValue ?? '' ) }
        />
    );
};
```

Props

disabled

If true, the input will be disabled.

- Type: Boolean
- Required: No
- Default: false

isPressEnterToChange

If true, the ENTER key press is required in order to trigger an onChange. If enabled, a change is also triggered when tabbing away (onBlur).

- Type: Boolean
- Required: No
- Default: false

hideLabelFromVision

If true, the label will only be visible to screen readers.

- Type: Boolean
- Required: No

label

If this property is added, a label will be generated using label property as the content.

- Type: String
- Required: No

labelPosition

The position of the label (top, side, bottom, or edge).

- Type: String
- Required: No

onChange

A function that receives the value of the input.

- Type: Function
- Required: Yes

prefix

Renders an element on the left side of the input.

- Type: React.ReactNode

- Required: No

[size](#)

Adjusts the size of the input.

Sizes include: `default`, `small`

- Type: `String`
- Required: No
- Default: `default`

[suffix](#)

Renders an element on the right side of the input.

- Type: `React.ReactNode`
- Required: No

[type](#)

Type of the input element to render. Defaults to “text”.

- Type: `String`
- Required: No
- Default: “text”

[value](#)

The current value of the input.

- Type: `String`
- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: InputControl](#)”

[Previous](#) [Icon Previous: Icon](#)

[Next IsolatedEventContainer](#) [Next: IsolatedEventContainer](#)

IsolatedEventContainer

In this article

Table of Contents

- [Usage](#)
- [Props](#)

[↑ Back to top](#)

Deprecated

This is a container that prevents certain events from propagating outside of the container. This is used to wrap UI elements such as modals and popovers where the propagated event can cause problems. The event continues to work inside the component.

For example, a `mousedown` event in a modal container can propagate to the surrounding DOM, causing UI outside of the modal to be interacted with.

The current isolated events are:

- `mousedown` – This prevents UI interaction with other `mousedown` event handlers, such as selection

[Usage](#)

Creates a custom component that won't propagate `mousedown` events outside of the component.

```
import { IsolatedEventContainer } from '@wordpress/components';

const MyModal = () => {
    return (
        <IsolatedEventContainer
            className="component-some_component"
            onClick={ clickHandler }
        >
            <p>This is an isolated component</p>
        </IsolatedEventContainer>
    );
};
```

[Props](#)

All props are passed as-is to the `<IsolatedEventContainer />`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: IsolatedEventContainer”](#)

[Previous InputControl](#) [Previous: InputControl](#)

[Next ItemGroup](#) [Next: ItemGroup](#)

ItemGroup

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [isBordered: boolean](#)
 - [isRounded: boolean](#)
 - [isSeparated: boolean](#)
 - [size: ‘small’ | ‘medium’ | ‘large’](#)
 - [Context](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

ItemGroup displays a list of Items grouped and styled together.

[Usage](#)

ItemGroup should be used in combination with the [Item sub-component](#).

```
import {  
    __experimentalItemGroup as ItemGroup,  
    __experimentalItem as Item,  
} from '@wordpress/components';  
  
function Example() {  
    return (  
        <ItemGroup>  
            <Item>Code</Item>  
            <Item>is</Item>  
            <Item>Poetry</Item>  
        </ItemGroup>  
    );  
}
```

```
) ;  
}
```

Props

isBordered: boolean

Renders borders around each items.

- Required: No
- Default: `false`

isRounded: boolean

Renders with rounded corners.

- Required: No
- Default: `true`

isSeparated: boolean

Renders items individually. Even if `isBordered` is `false`, a border in between each item will be still be displayed.

- Required: No
- Default: `false`

size: 'small' | 'medium' | 'large'

Determines the amount of padding within the component.

When not defined, it defaults to the value from the context (which is `medium` by default).

- Required: No
- Default: `medium`

Context

The [Item sub-component](#) is connected to `<ItemGroup />` using [Context](#). Therefore, `Item` receives the `size` prop from the `ItemGroup` parent component.

In the following example, the `<Item />` will render with a size of `small`:

```
import {  
    __experimentalItemGroup as ItemGroup,  
    __experimentalItem as Item,  
} from '@wordpress/components';  
  
const Example = () => (  
    <ItemGroup size="small">  
        <Item>Item text</Item>  
    </ItemGroup>  
) ;
```

First published

July 29, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ItemGroup](#)

[Previous IsolatedEventContainer](#) [Previous: IsolatedEventContainer](#)

[Next Item](#) [Next: Item](#)

Item

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [onClick: React.MouseEventHandler<HTMLDivElement>](#)
 - [size: ‘small’ | ‘medium’ | ‘large’](#)
 - [Context](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Item is used in combination with ItemGroup to display a list of items grouped and styled together.

Usage

Item should be used in combination with the [ItemGroup component](#).

```
import {  
  __experimentalItemGroup as ItemGroup,  
  __experimentalItem as Item,  
} from '@wordpress/components';  
  
function Example() {  
  return (  
    <ItemGroup>  
      <Item>Code</Item>  
      <Item>is</Item>  
      <Item>Poetry</Item>  
    </ItemGroup>  
  );  
}
```

```
) ;  
}
```

Props

[onClick: React.MouseEventHandler<HTMLDivElement>](#)

Even handler for processing `click` events. When defined, the `Item` component will render as a button (unless differently specified via the `as` prop).

- Required: No

[size: ‘small’ | ‘medium’ | ‘large’](#)

Determines the amount of padding within the component.

- Required: No
- Default: `medium`

Context

`Item` is connected to [the `<ItemGroup />` parent component](#) using [Context](#). Therefore, `Item` receives the `size` prop from the `ItemGroup` parent component.

In the following example, the `<Item />` will render with a size of `small`:

```
import {  
    __experimentalItemGroup as ItemGroup,  
    __experimentalItem as Item,  
} from '@wordpress/components';  
  
const Example = () => (  
    <ItemGroup size="small">  
        <Item>...</Item>  
    </ItemGroup>  
) ;
```

First published

July 29, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Item”](#)

[Previous ItemGroup](#) [Previous: ItemGroup](#)

[Next KeyboardShortcuts](#) [Next: KeyboardShortcuts](#)

KeyboardShortcuts

In this article

Table of Contents

- [Example](#)
- [Props](#)
 - [children](#)
 - [shortcuts](#)
 - [bindGlobal](#)
 - [eventName](#)
- [References](#)

[↑ Back to top](#)

<KeyboardShortcuts /> is a component which handles keyboard sequences during the lifetime of the rendering element.

When passed children, it will capture key events which occur on or within the children. If no children are passed, events are captured on the document.

It uses the [Mousetrap](#) library to implement keyboard sequence bindings.

[Example](#)

Render <KeyboardShortcuts /> with a `shortcuts` prop object:

```
import { useState } from 'react';
import { KeyboardShortcuts } from '@wordpress/components';

const MyKeyboardShortcuts = () => {
    const [ isSelected, setIsAllSelected ] = useState( false );
    const selectAll = () => {
        setIsAllSelected( true );
    };

    return (
        <div>
            <KeyboardShortcuts
                shortcuts={ {
                    'mod+a': selectAll,
                } }
            />
            [cmd/ctrl + A] Combination pressed? { isSelected ? 'Yes' :
        </div>
    );
}
```

Props

The component accepts the following props:

children

Elements to render, upon whom key events are to be monitored.

- Type: `ReactNode`
- Required: No

shortcuts

An object of shortcut bindings, where each key is a keyboard combination, the value of which is the callback to be invoked when the key combination is pressed.

- Type: `Object`
- Required: Yes

Note: The value of each shortcut should be a consistent function reference, not an anonymous function. Otherwise, the callback will not be correctly unbound when the component unmounts.

Note: The `KeyboardShortcuts` component will not update to reflect a changed `shortcuts` prop. If you need to change shortcuts, mount a separate `KeyboardShortcuts` element, which can be achieved by assigning a unique `key` prop.

bindGlobal

By default, a callback will not be invoked if the key combination occurs in an editable field. Pass `bindGlobal` as `true` if the key events should be observed globally, including within editable fields.

- Type: `Boolean`
- Required: No

Tip: If you need some but not all keyboard events to be observed globally, simply render two distinct `KeyboardShortcuts` elements, one with and one without the `bindGlobal` prop.

eventName

By default, a callback is invoked in response to the `keydown` event. To override this, pass `eventName` with the name of a specific keyboard event.

- Type: `String`
- Required: No

References

- [Mousetrap documentation](#)

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: KeyboardShortcuts”](#)

[Previous Item](#) [Previous: Item](#)

[Next MenuGroup](#) [Next: MenuGroup](#)

MenuGroup

In this article

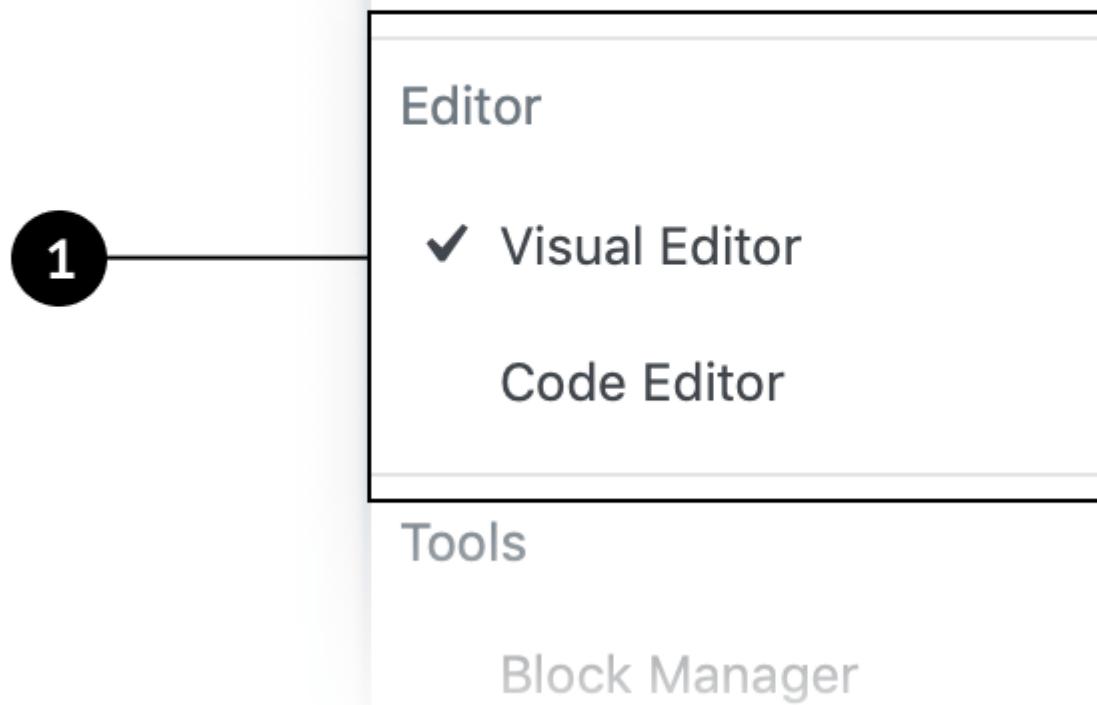
[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
- [Development guidelines](#)
 - [Usage](#)
- [Related Components](#)

[↑ Back to top](#)

MenuGroup wraps a series of related MenuItem components into a common section.

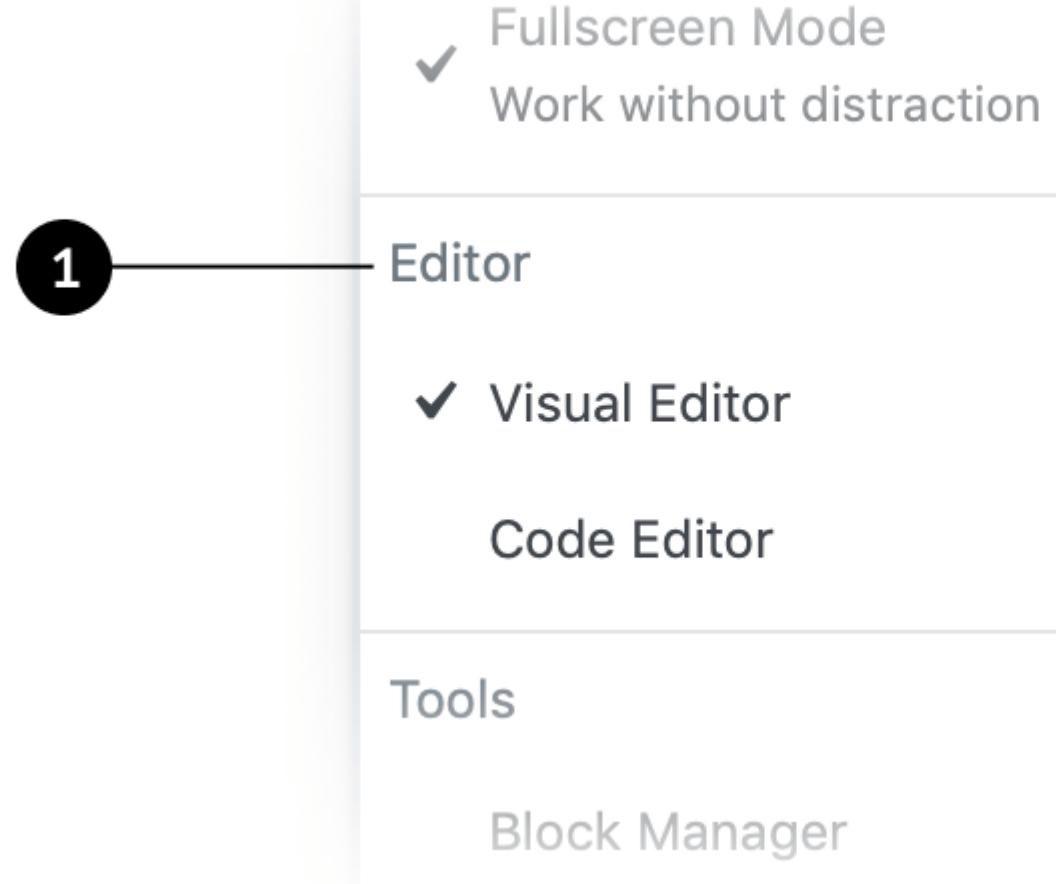
✓ Fullscreen Mode
Work without distraction



Design guidelines

Usage

A **MenuGroup** should be used to indicate that two or more individual **MenuItem**s are related. When other menu items exist above or below a **MenuGroup**, the group should have a divider line between it and the adjacent item. A **MenuGroup** can optionally include a label to describe its contents.



1. MenuGroup label
2. MenuGroup dividers

Development guidelines

Usage

```
import { MenuGroup, MenuItem } from '@wordpress/components';

const MyMenuGroup = () => (
    <MenuGroup label="Settings">
        <MenuItem>Setting 1</MenuItem>
        <MenuItem>Setting 2</MenuItem>
    </MenuGroup>
);
```

Related Components

- MenuGroups are intended to be used in a DropDownMenu.
- To use a single button in a menu, use MenuItem.
- To allow users to toggle between a set of menu options, use MenuItemChoice inside of a MenuGroup.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: MenuGroup](#)

[Previous KeyboardShortcuts](#) [Previous: KeyboardShortcuts](#)

[Next MenuItem](#) [Next: MenuItem](#)

MenuItem

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [children](#)
 - [disabled](#)
 - [info](#)
 - [icon](#)
 - [iconPosition](#)
 - [isSelected](#)
 - [shortcut](#)
 - [role](#)
 - [suffix](#)

[↑ Back to top](#)

MenuItem is a component which renders a button intended to be used in combination with the [DropdownMenu component](#).

Usage

```
import { useState } from 'react';
import { MenuItem } from '@wordpress/components';

const MyMenuItem = () => {
    const [ isActive, setIsActive ] = useState( true );

    return (
        <MenuItem
            icon={ isActive ? 'yes' : 'no' }
            isSelected={ isActive }
            onClick={ () => setIsActive( ( state ) => ! state ) }
    )
}
```

```
>
    Toggle
  </MenuItem>
);
}
```

Props

MenuItem supports the following props. Any additional props are passed through to the underlying [Button](#).

children

- Type: `Element`
- Required: No

Element to render as child of button.

disabled

- Type: `boolean`
- Required: No

Refer to documentation for [Button's disabled prop](#).

info

- Type: `string`
- Required: No

Text to use as description for button text.

Refer to documentation for [label](#).

icon

- Type: `string`
- Required: No

Refer to documentation for [Button's icon prop](#).

iconPosition

- Type: `string`
- Required: No
- Default: 'right'

Determines where to display the provided icon.

isSelected

- Type: `boolean`

- Required: No

Whether or not the menu item is currently selected. `isSelected` is only taken into account when the `role` prop is either "`menuItemcheckbox`" or "`menuItemradio`".

[shortcut](#)

- Type: `string` or `object`
- Required: No

If `shortcut` is a string, it is expecting the display text. If `shortcut` is an object, it will accept the properties of `display` (`string`) and `ariaLabel` (`string`).

[role](#)

- Type: `string`
- Required: No
- Default: '`menuItem`'

[Aria Spec](#). If you need to have selectable menu items use `menuItemradio` for single select, and `menuItemcheckbox` for multiselect.

[suffix](#)

- Type: `Element`
- Required: No

Allows for markup other than icons or shortcuts to be added to the menu item.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: MenuItem](#)

[Previous MenuGroup](#) [Previous: MenuGroup](#)

[Next MenuItemsChoice](#) [Next: MenuItemsChoice](#)

MenuItemsChoice

In this article

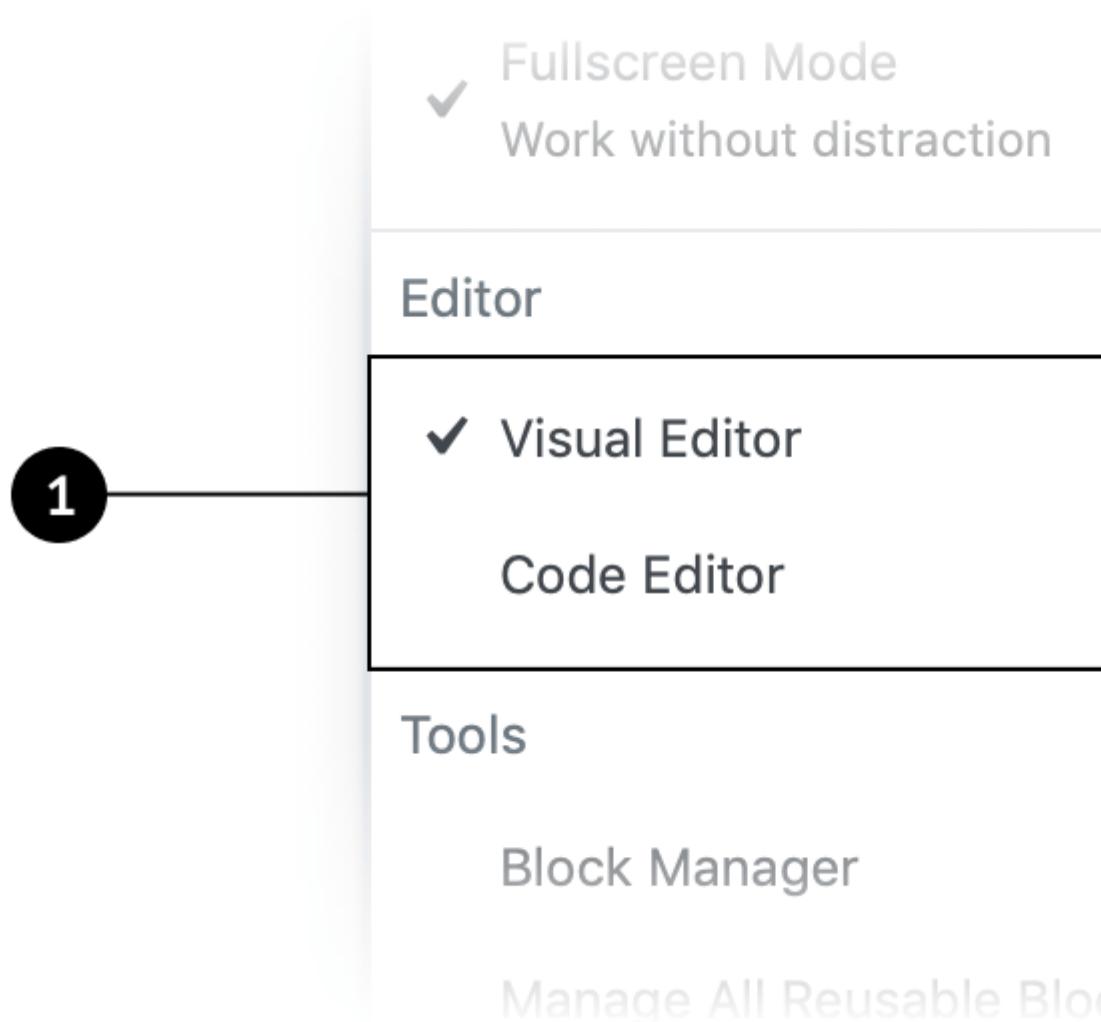
Table of Contents

- [Design guidelines](#)
 - [Usage](#)

- [Development guidelines](#)
 - [Usage](#)

[↑ Back to top](#)

MenuItemChoice functions similarly to a set of MenuItem, but allows the user to select one option from a set of multiple choices.



[Design guidelines](#)

A MenuItemChoice should be housed within its own distinct MenuGroup, so that the set of options are distinct from nearby MenuItem.

[Usage](#)

MenuItemChoice is used in a DropdownMenu to present users with a set of options. When one option in a MenuItemChoice is selected, the others are automatically deselected.

✓ Fullscreen Mode
Work without distraction

Editor

1

✓ Visual Editor

Code Editor

Tools

Block Manager

Manage All Reusable Blo

1. A checkmark icon appears next to the choice when it's enabled, and disappears when disabled.
2. If an item in `MenuItemChoice` has an associated keyboard shortcut, that should be displayed to the right of the menu title, aligned to the right side of the menu item. Selected choices should not have visible shortcuts, since they are already active.

When to use `MenuItemChoice`

Use `MenuItemChoice` when you want users to:

- Select a single option from a set of choices in a menu.
- Expose all available options.

`MenuItemChoice` should not be used to toggle individual features on and off. For that, consider using a `FeatureToggle`.

Defaults

When using `MenuItemChoice`, **one option should be selected by default** (i.e., when the page loads, in the case of a web application).

User control

Selecting an option by default communicates that the user is required to choose one in the set.

Expediting tasks

When one choice in a set of `MenuItemsChoice` is the most desirable or frequently selected, it's helpful to select it by default. Doing this reduces the interaction cost and can save the user time and clicks.

The power of suggestion

Designs with a `MenuItemsChoice` option selected by default make a strong suggestion to the user. It can help them make the best decision and increase their confidence. (Use this guidance with caution, and only for good.)

Development guidelines

Usage

```
import { useState } from 'react';
import { MenuGroup, MenuItemsChoice } from '@wordpress/components';

const MyMenuItemsChoice = () => {
    const [ mode, setMode ] = useState( 'visual' );
    const choices = [
        {
            value: 'visual',
            label: 'Visual editor',
        },
        {
            value: 'text',
            label: 'Code editor',
        },
    ];
    return (
        <MenuGroup label="Editor">
            <MenuItemsChoice
                choices={ choices }
                value={ mode }
                onSelect={ ( newMode ) => setMode( newMode ) }
            />
        </MenuGroup>
    );
};
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: `MenuItemsChoice`](#)

[Previous MenuItem](#) [Previous: MenuItem](#)

[Next Modal](#) [Next: Modal](#)

Modal

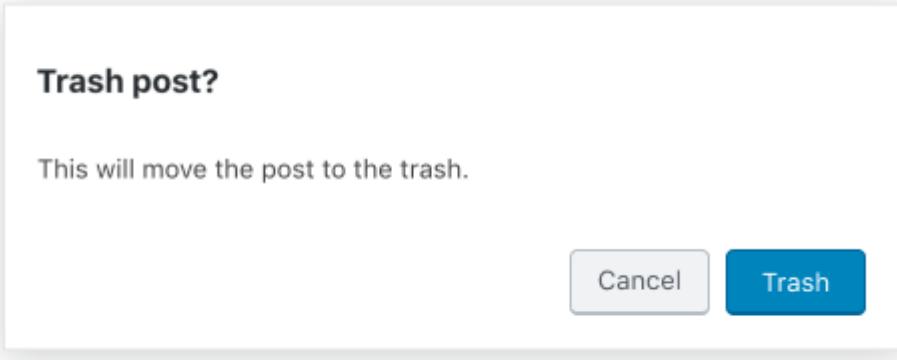
In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
 - [Anatomy](#)
 - [Modal box and scrim](#)
 - [Title](#)
 - [Buttons](#)
 - [Behavior](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

Modals give users information and choices related to a task they’re trying to accomplish. They can contain critical information, require decisions, or involve multiple tasks.



Trash post?

This will move the post to the trash.

[Cancel](#)

[Trash](#)

Design guidelines

Usage

A modal is a type of floating window that appears in front of content to provide critical information or ask for a decision. Modals disable all other functionality when they appear. A modal remains on screen until the user confirms it, dismisses it, or takes the required action.

While modals can be an effective way to disclose additional controls or information, they can also be a source of interruption for the user. For this reason, always question whether a modal is necessary, and work to avoid the situations in which they are required.

Principles

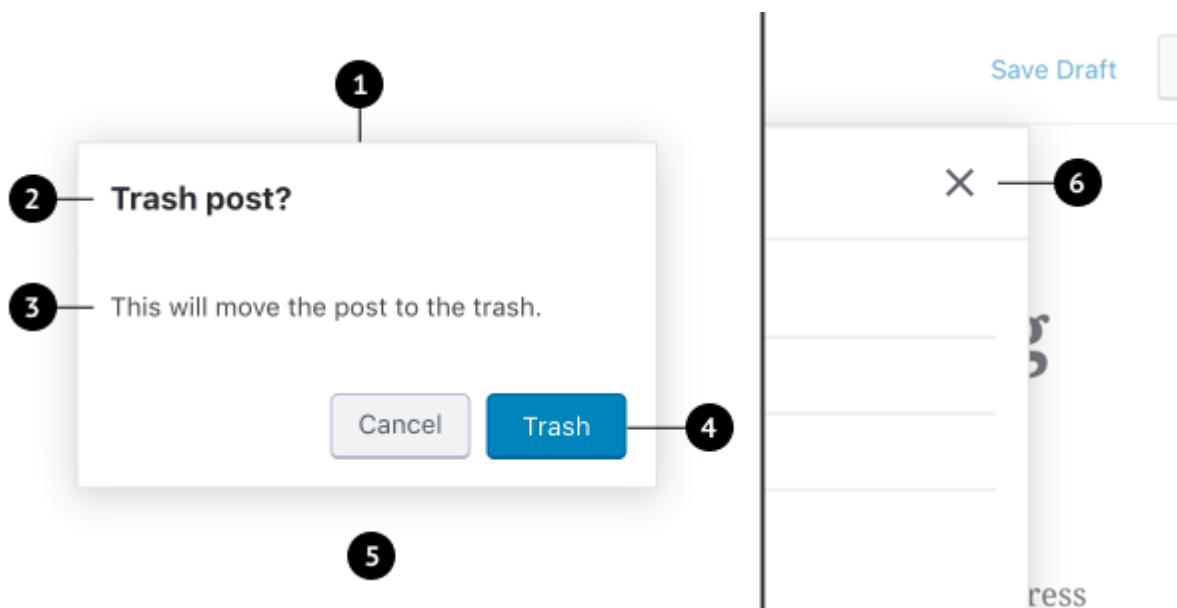
- **Focused.** Modals pull user attention away from the rest of the screen to focus their attention, ensuring that the modal's content is addressed.
- **Direct.** Modal text should communicate important information and be dedicated to helping the user appropriately complete a task.
- **Helpful.** Modals should appear in response to a user task or an action to offer relevant and contextual information.

When to use

Modals are used for:

- Errors that block normal operation.
- Critical information that requires a specific user task, decision, or acknowledgement.
- Contextual information that appears in response to a user task or action.

Anatomy



1. Container
2. Title
3. Supporting text
4. Buttons

5. Scrim
6. Close button (optional)

Modal box and scrim

A modal is a type of window. Access to the rest of the UI is disabled until the modal is addressed. All modals are interruptive by design – their purpose is to have the user focus on content, so the modal surface appears in front of all other surfaces.

To clarify that the rest of the screen is inaccessible and to focus attention on the modal, surfaces behind the modal are scrimmed — they get a temporary overlay to obscure their content and make it less prominent.

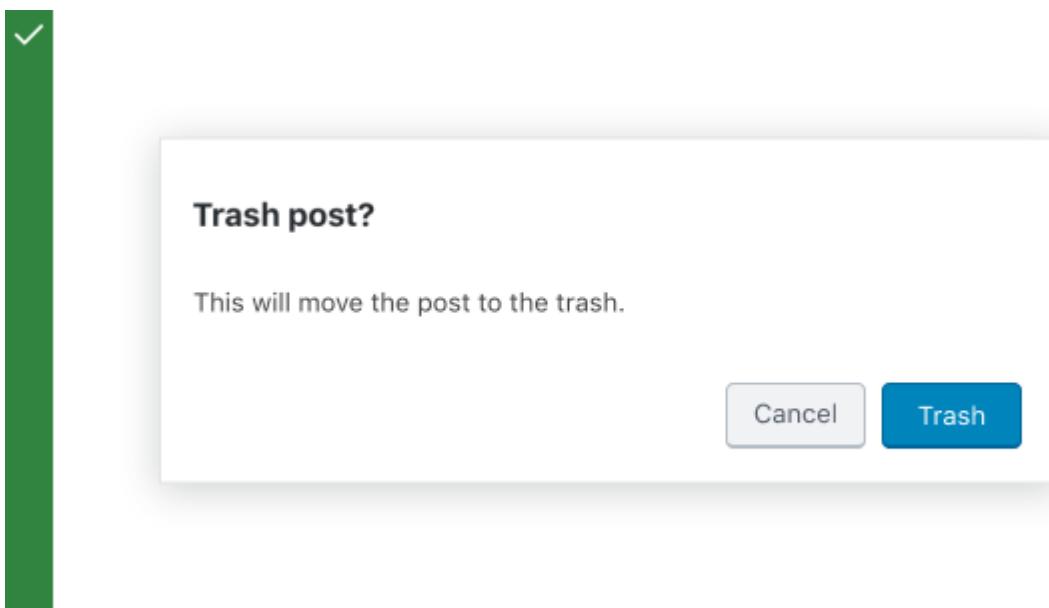
Title

A modal's purpose is communicated through its title and button text.

All modals should have a title for accessibility reasons (the `contentLabel` prop can be used to set titles that aren't visible).

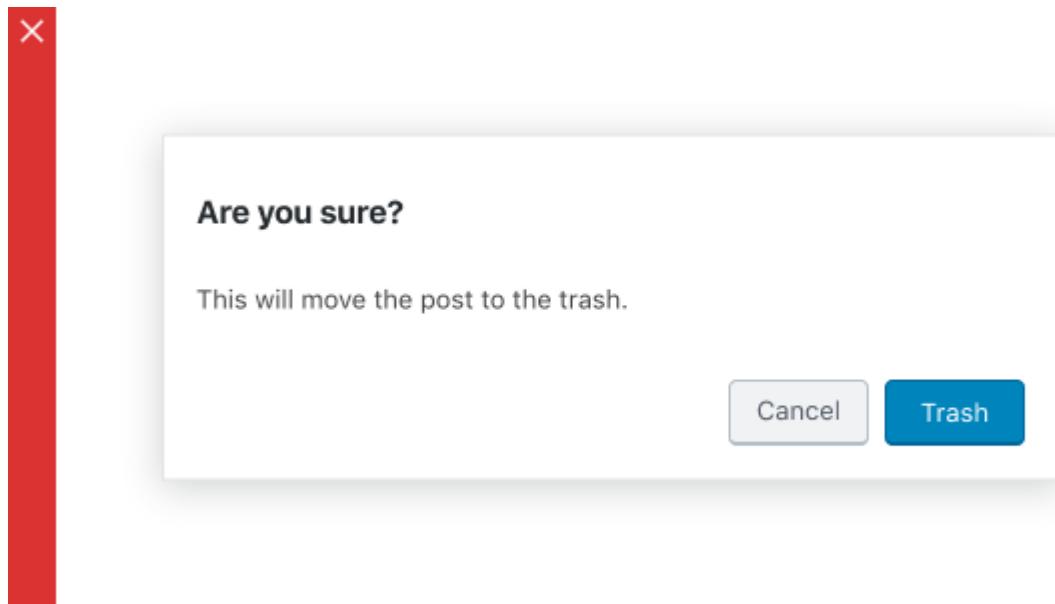
Titles should:

- Contain a brief, clear statement or question
- Avoid apologies (“Sorry for the interruption”), alarm (“Warning!”), or ambiguity (“Are you sure?”).



Do

This modal title poses a specific question, concisely explains the purpose the request, and provides clear actions.



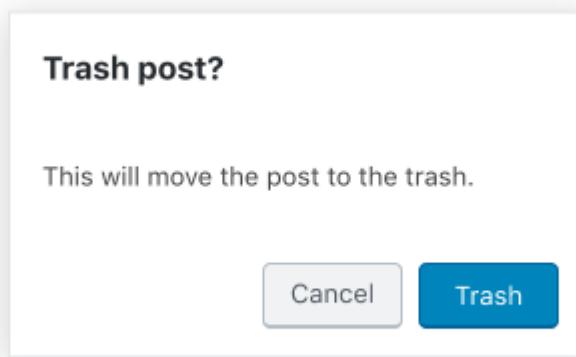
Don't

This modal creates ambiguity, and therefore unease — it leaves the user unsure about how to respond, or causes them to second-guess their answer.

Buttons

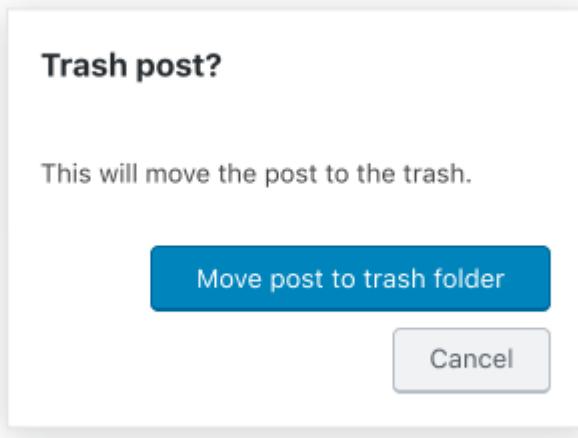
Side-by-side buttons (recommended)

Side-by-side buttons display two text buttons next to one another.



Stacked or full-width buttons

Use stacked buttons when you need to accommodate longer button text. Always place confirming actions above dismissive actions.



Behavior

Modals appear without warning, requiring users to stop their current task. They should be used sparingly — not every choice or setting warrants this kind of abrupt interruption.

Position

Modals retain focus until dismissed or the user completes an action, like choosing a setting. They shouldn't be obscured by other elements or appear partially on screen.

Scrolling

Most modal content should avoid scrolling. Scrolling is permissible if the modal content exceeds the height of the modal (e.g. a list component with many rows). When a modal scrolls, the modal title is pinned at the top and the buttons are pinned at the bottom. This ensures that content remains visible alongside the title and buttons, even while scrolling.

Modals don't scroll with elements outside of the modal, like the background.

When viewing a scrollable list of options, the modal title and buttons remain fixed.

Dismissing modals

Modals are dismissible in three ways:

- Tapping outside of the modal
- Tapping the “Cancel” button
- Tapping the “Close” icon button, or pressing the `esc` key

If the user's ability to dismiss a modal is disabled, they must choose a modal action to proceed.

Development guidelines

The modal is used to create an accessible modal over an application.

Note: The API for this modal has been mimicked to resemble [react-modal](#).

Usage

The following example shows you how to properly implement a modal. For the modal to properly work it's important you implement the close logic for the modal properly.

```
import { useState } from 'react';
import { Button, Modal } from '@wordpress/components';

const MyModal = () => {
    const [ isOpen, setOpen ] = useState( false );
    const openModal = () => setOpen( true );
    const closeModal = () => setOpen( false );

    return (
        <>
            <Button variant="secondary" onClick={ openModal }>
                Open Modal
            </Button>
            { isOpen && (
                <Modal title="This is my modal" onRequestClose={ closeModal }>
                    <Button variant="secondary" onClick={ closeModal }>
                        My custom close button
                    </Button>
                </Modal>
            ) }
        </>
    );
};


```

Props

The set of props accepted by the component will be specified below.
Props not included in this set will be applied to the input elements.

aria.describedby: string

If this property is added, it will be added to the modal content div as `aria-describedby`.

- Required: No

aria.labelledby: string

If this property is added, it will be added to the modal content div as `aria-labelledby`.
Use this when you are rendering the title yourself within the modal's content area instead of using the `title` prop. This ensures the title is usable by assistive technology.

Titles are required for accessibility reasons, see `contentLabel` and `title` for other ways to provide a title.

- Required: No

- Default: if the `title` prop is provided, this will default to the id of the element that renders `title`

`bodyOpenClassName: string`

Class name added to the body element when the modal is open.

- Required: No
- Default: `modal-open`

`className: string`

If this property is added, it will add an additional class name to the modal content `div`.

- Required: No

`contentLabel: string`

If this property is added, it will be added to the modal content `div` as `aria-label`.

Titles are required for accessibility reasons, see `aria-labelledby` and `title` for other ways to provide a title.

- Required: No

`focusOnMount: boolean | 'firstElement' | 'firstContentElement'`

If this property is true, it will focus the first tabbable element rendered in the modal.

If this property is false, focus will not be transferred and it is the responsibility of the consumer to ensure accessible focus management.

If set to `firstElement` focus will be placed on the first tabbable element anywhere within the Modal.

If set to `firstContentElement` focus will be placed on the first tabbable element within the Modal's **content** (i.e. children). Note that it is the responsibility of the consumer to ensure there is at least one tabbable element within the children **or the focus will be lost**.

- Required: No
- Default: `true`

`headerActions`

An optional React node intended to contain additional actions or other elements related to the modal, for example, buttons. Content is rendered in the top right corner of the modal and to the left of the close button, if visible.

- Required: No
- Default: `null`

`isDismissible: boolean`

If this property is set to false, the modal will not display a close icon and cannot be dismissed.

- Required: No
- Default: `true`

`isFullScreen: boolean`

This property when set to `true` will render a full screen modal.

- Required: No
- Default: `false`

`size: 'small' | 'medium' | 'large' | 'fill'`

If this property is added it will cause the modal to render at a preset width, or expand to fill the screen. This prop will be ignored if `isFullScreen` is set to `true`.

- Required: No

Note: Modal's width can also be controlled by adjusting the width of the modal's contents via CSS.

`onRequestClose: “`

This function is called to indicate that the modal should be closed.

- Required: Yes

`overlayClassName: string`

If this property is added, it will add an additional class name to the modal overlay div.

- Required: No

`role: AriaRole`

If this property is added, it will override the default role of the modal.

- Required: No
- Default: `dialog`

`shouldCloseOnClickOutside: boolean`

If this property is added, it will determine whether the modal requests to close when a mouse click occurs outside of the modal content.

- Required: No
- Default: `true`

`shouldCloseOnEsc: boolean`

If this property is added, it will determine whether the modal requests to close when the escape key is pressed.

- Required: No
- Default: `true`

`style: CSSProperties`

If this property is added, it will be added to the modal frame `div`.

- Required: No

`title: string`

This property is used as the modal header's title.

Titles are required for accessibility reasons, see `aria-labelledby` and `contentLabel` for other ways to provide a title.

- Required: No

`__experimentalHideHeader: boolean`

When set to `true`, the Modal's header (including the icon, title and close button) will not be rendered.

Warning: This property is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

- Required: No
- Default: `false`

Related components

- To notify a user with a message of medium importance, use `Notice`.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Modal](#)

[Previous](#) [MenuItemsChoice](#) [Previous: MenuItemsChoice](#)

[Next](#) [NavigableContainer](#) [Next: NavigableContainer](#)

NavigableContainer

In this article

[Table of Contents](#)

- [Props](#)
 - [cycle: boolean](#)
 - [eventToOffset: \(event: KeyboardEvent \) => -1 | 0 | 1 | undefined](#)
 - [onKeyDown: \(event: KeyboardEvent \) => void](#)
 - [onNavigate: \(index: number, focusable: HTMLElement \) => void](#)
 - [orientation: 'vertical' | 'horizontal' | 'both'](#)
- [Components](#)
 - [NavigableMenu](#)
 - [TabbableContainer](#)
 - [Usage](#)

[↑ Back to top](#)

`NavigableContainer` is a React component to render a container navigable using the keyboard. Only things that are focusable can be navigated to. It will currently always be a `div`.

`NavigableContainer` is exported as two components: `NavigableMenu` and `TabbableContainer`. `NavigableContainer` itself is **not** exported. `NavigableMenu` and `TabbableContainer` have the props listed below. Any other props will be passed through to the `div`.

[Props](#)

These are the props that `NavigableMenu` and `TabbableContainer`. Any props which are specific to one component are labelled appropriately.

[cycle: boolean](#)

A boolean which tells the component whether or not to cycle from the end back to the beginning and vice versa.

- Required: No
- default: `true`

[eventToOffset: \(event: KeyboardEvent \) => -1 | 0 | 1 | undefined](#)

(`TabbableContainer` only)
Gets an offset, given an event.

- Required: No

onKeyDown: (event: KeyboardEvent)=> void

A callback invoked on the keydown event.

- Required: No

onNavigate: (index: number, focusable: HTMLElement)=> void

A callback invoked when the menu navigates to one of its children passing the index and child as an argument

- Required: No

orientation: 'vertical' | 'horizontal' | 'both'

(NavigableMenu only)

The orientation of the menu. It could be “vertical”, “horizontal”, or “both”.

- Required: No
- Default: "vertical"

Components

NavigableMenu

A NavigableMenu allows movement up and down (or left and right) the component via the arrow keys. The tab key is not handled. The `orientation` prop is used to determine whether the arrow keys used are vertical, horizontal or both.

The NavigableMenu by default has a `menu` role and therefore, in order to function as expected, the component expects its children elements to have one of the following roles: '`MenuItem`' | '`MenuItemRadio`' | '`MenuItemCheckbox`'.

TabbableContainer

A TabbableContainer will only be navigated using the tab key. Every intended tabstop must have a tabIndex 0.

Usage

```
import {
  NavigableMenu,
  TabbableContainer,
  Button,
} from '@wordpress/components';

function onNavigate( index, target ) {
  console.log( `Navigates to ${ index }`, target );
}

const MyNavigableContainer = () => (
  <div>
```

```
<span>Navigable Menu:</span>
<NavigableMenu onNavigate={ onNavigate } orientation="horizontal">
  <Button variant="secondary">Item 1</Button>
  <Button variant="secondary">Item 2</Button>
  <Button variant="secondary">Item 3</Button>
</NavigableMenu>

<span>Tabbable Container:</span>
<TabbableContainer onNavigate={ onNavigate }>
  <Button variant="secondary" tabIndex="0">
    Section 1
  </Button>
  <Button variant="secondary" tabIndex="0">
    Section 2
  </Button>
  <Button variant="secondary" tabIndex="0">
    Section 3
  </Button>
  <Button variant="secondary" tabIndex="0">
    Section 4
  </Button>
</TabbableContainer>
</div>
);


```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NavigableContainer](#)

[Previous Modal](#) [Previous: Modal](#)
[Next Navigation](#) [Next: Navigation](#)

Navigation

In this article

[Table of Contents](#)

- [Usage](#)
- [Navigation Props](#)
 - [activeItem](#)
 - [activeMenu](#)
 - [className](#)
 - [onActivateMenu](#)

- [Navigation Menu Props](#)
 - [backButtonLabel](#)
 - [onBackButtonClick](#)
 - [className](#)
 - [hasSearch](#)
 - [menu](#)
 - [onSearch](#)
 - [isSearchDebouncing](#)
 - [parentMenu](#)
 - [search](#)
 - [isEmpty](#)
 - [title](#)
 - [titleAction](#)
- [Navigation Group Props](#)
 - [className](#)
 - [title](#)
- [Navigation Item Props](#)
 - [badge](#)
 - [className](#)
 - [href](#)
 - [icon](#)
 - [item](#)
 - [navigateToMenu](#)
 - [hideIfTargetMenuEmpty](#)
 - [onClick](#)
 - [isText](#)
 - [title](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Render a navigation list with optional groupings and hierarchy.

[Usage](#)

```
import {
  __experimentalNavigation as Navigation,
  __experimentalNavigationGroup as NavigationGroup,
  __experimentalNavigationItem as NavigationItem,
  __experimentalNavigationMenu as NavigationMenu,
} from '@wordpress/components';

const MyNavigation = () => (
  <Navigation>
    <NavigationMenu title="Home">
      <NavigationGroup title="Group 1">
        <NavigationItem item="item-1" title="Item 1" />
        <NavigationItem item="item-2" title="Item 2" />
      </NavigationGroup>
      <NavigationGroup title="Group 2">
```

```

        <NavigationItem
            item="item-3"
            navigateToMenu="category"
            title="Category"
        />
    </NavigationGroup>
</NavigationMenu>

<NavigationMenu
    backButtonLabel="Home"
    menu="category"
    parentMenu="root"
    title="Category"
>
    <NavigationItem badge="1" item="child-1" title="Child 1" />
    <NavigationItem item="child-2" title="Child 2" />
</NavigationMenu>
</Navigation>
);

```

Navigation Props

Navigation supports the following props.

activeItem

- Type: `string`
- Required: No

The active item slug.

activeMenu

- Type: `string`
- Required: No
- Default: “root”

The active menu slug.

className

- Type: `string`
- Required: No

Optional className for the Navigation component.

onActivateMenu

- Type: `function`
- Required: No

Sync the active menu between the external state and the Navigation’s internal state.

Navigation Menu Props

NavigationMenu supports the following props.

backButtonLabel

- Type: `string`
- Required: No
- Default: parent menu's title or “Back”

The back button label used in nested menus. If not provided, the label will be inferred from the parent menu's title.

If for some reason the parent menu's title is not available then it will default to “Back”.

onBackButtonClick

- Type: `function`
- Required: No

A callback to handle clicking on the back button. If this prop is provided then the back button will be shown.

className

- Type: `string`
- Required: No

Optional className for the NavigationMenu component.

hasSearch

- Type: `boolean`
- Required: No

Enable the search feature on the menu title.

menu

- Type: `string`
- Required: No
- Default: “root”

The unique identifier of the menu. The root menu can omit this, and it will default to “root”; all other menus need to specify it.

onSearch

- Type: `(searchString: string) => void;`
- Required: No

When `hasSearch` is active, this function handles the search input's `onChange` event, making it controlled from the outside. It requires setting the `search` prop as well.

[isSearchDebouncing](#)

- Type: `boolean`
- Required: No

Indicates whether the search is debouncing or not. In case of `true` the “No results found.” text is omitted. Used to prevent showing “No results found.” text between debounced searches.

[parentMenu](#)

- Type: `string`
- Required: No

The parent menu slug; used by nested menus to indicate their parent menu.

[search](#)

- Type: `string`
- Required: No

When `hasSearch` is active and `onSearch` is provided, this controls the value of the search input. Required when the `onSearch` prop is provided.

[isEmpty](#)

- Type: `boolean`
- Required: No

Indicates whether the menu is empty or not. Used together with the `hideIfTargetMenuEmpty` prop of Navigation Item.

[title](#)

- Type: `string`
- Required: No

The menu title. It’s also the field used by the menu search function.

[titleAction](#)

- Type: `React.ReactNode`
- Required: No

Use this prop to render additional actions in the menu title.

[Navigation Group Props](#)

`NavigationGroup` supports the following props.

[className](#)

- Type: `string`

- Required: No

Optional className for the `NavigationGroup` component.

[title](#)

- Type: `string`
- Required: No

The group title.

[Navigation Item Props](#)

`NavigationItem` supports the following props.

[badge](#)

- Type: `string | Number`
- Required: No

The item badge content.

[className](#)

- Type: `string`
- Required: No

Optional className for the `NavigationItem` component.

[href](#)

- Type: `string`
- Required: No

If provided, renders `a` instead of `button`.

[icon](#)

- Type: `JSX.Element`
- Required: No

If no `children` are passed, this prop allows to specify a custom icon for the menu item.

[item](#)

- Type: `string`
- Required: No

The unique identifier of the item.

[navigateToMenu](#)

- Type: `string`
- Required: No

The child menu slug. If provided, clicking on the item will navigate to the target menu.

[hideIfTargetMenuEmpty](#)

- Type: `boolean`
- Required: No

Indicates whether this item should be hidden if the menu specified in `navigateToMenu` is marked as empty in the `isEmpty` prop. Used together with the `isEmpty` prop of Navigation Menu.

[onClick](#)

- Type: `React.MouseEventHandler`
- Required: No

A callback to handle clicking on a menu item.

[isText](#)

- Type: `boolean`
- Required: No
- Default: `false`

If set to true then the menu item will only act as a text-only item rather than a button.

[title](#)

- Type: `string`
- Required: No

The item title.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Navigation”](#)

[Previous NavigableContainer](#) [Previous: NavigableContainer](#)
[Next NavigatorBackButton](#) [Next: NavigatorBackButton](#)

NavigatorBackButton

In this article

Table of Contents

- [Usage](#)
 - [Inherited props](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

The `NavigatorBackButton` component can be used to navigate to a screen and should be used in combination with the [NavigatorProvider](#), the [NavigatorScreen](#) and the [NavigatorButton](#) components (or the `useNavigator` hook).

[Usage](#)

Refer to [the NavigatorProvider component](#) for a usage example.

[Inherited props](#)

`NavigatorBackButton` also inherits all of the [Button props](#), except for `href` and `target`.

First published

February 17, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NavigatorBackButton](#)

[Previous Navigation](#) [Previous: Navigation](#)
[Next NavigatorButton](#) [Next: NavigatorButton](#)

NavigatorButton

In this article

Table of Contents

- [Usage](#)

- [Props](#)

- [attributeName: string](#)
- [onClick: React.MouseEventHandler< HTMLElement >](#)
- [path: string](#)
- [Inherited props](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

The `NavigatorButton` component can be used to navigate to a screen and should be used in combination with the [NavigatorProvider](#), the [NavigatorScreen](#) and the [NavigatorBackButton](#) components (or the `useNavigator` hook).

[Usage](#)

Refer to [the NavigatorProvider component](#) for a usage example.

[Props](#)

The component accepts the following props:

[attributeName: string](#)

The HTML attribute used to identify the `NavigatorButton`, which is used by `Navigator` to restore focus.

- Required: No
- Default: `id`

[onClick: React.MouseEventHandler< HTMLElement >](#)

The callback called in response to a `click` event.

- Required: No

[path: string](#)

The path of the screen to navigate to. The value of this prop needs to be [a valid value for an HTML attribute](#).

- Required: Yes

[Inherited props](#)

`NavigatorButton` also inherits all of the [Button props](#), except for `href` and `target`.

First published

February 17, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NavigatorButton”](#)

[Previous NavigatorBackButton](#) [Previous: NavigatorBackButton](#)

[Next NavigatorProvider](#) [Next: NavigatorProvider](#)

NavigatorProvider

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [initialPath: string](#)
- [The navigator object](#)
 - [goTo: \(path: string, options: NavigateOptions \) => void](#)
 - [goToParent: \(\) => void;](#)
 - [goBack: \(\) => void](#)
 - [location: NavigatorLocation](#)
 - [params: Record< string, string | string\[\] >](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

The `NavigatorProvider` component allows rendering nested views/panels/menus (via the [NavigatorScreen component](#)) and navigate between these different states (via the [NavigatorButton](#), [NavigatorToParentButton](#) and [NavigatorBackButton](#) components or the `useNavigator` hook). The Global Styles sidebar is an example of this.

[Usage](#)

```
import {  
  __experimentalNavigatorProvider as NavigatorProvider,  
  __experimentalNavigatorScreen as NavigatorScreen,  
  __experimentalNavigatorButton as NavigatorButton,  
  __experimentalNavigatorToParentButton as NavigatorToParentButton,  
} from '@wordpress/components';  
  
const MyNavigation = () => (  
  <NavigatorProvider initialPath="/">  
    <NavigatorScreen path="/">  
      <p>This is the home screen.</p>
```

```

        <NavigatorButton path="/child">
            Navigate to child screen.
        </NavigatorButton>
    </NavigatorScreen>

    <NavigatorScreen path="/child">
        <p>This is the child screen.</p>
        <NavigatorToParentButton>
            Go back
        </NavigatorToParentButton>
    </NavigatorScreen>
</NavigatorProvider>
);

```

Important note

Parent/child navigation only works if the path you define are hierarchical, following a URL-like scheme where each path segment is separated by the / character.

For example:

- / is the root of all paths. There should always be a screen with path="/" .
- /parent/child is a child of /parent .
- /parent/child/grand-child is a child of /parent/child .
- /parent/:param is a child of /parent as well.

Props

The component accepts the following props:

initialPath: string

The initial active path.

- Required: No

The navigator object

You can retrieve a `navigator` instance by using the `useNavigator` hook.

The `navigator` instance has a few properties:

goTo: (path: string, options: NavigateOptions) => void

The `goTo` function allows navigating to a given path. The second argument can augment the navigation operations with different options.

The available options are:

- `focusTargetSelector: string`. An optional property used to specify the CSS selector used to restore focus on the matching element when navigating back.
- `isBack: boolean`. An optional property used to specify whether the navigation should be considered as backwards (thus enabling focus restoration when possible, and causing the animation to be backwards too)

[goToParent: \(\) => void;](#)

The `goToParent` function allows navigating to the parent screen.

Parent/child navigation only works if the path you define are hierarchical (see note above).

When a match is not found, the function will try to recursively navigate the path hierarchy until a matching screen (or the root `/`) are found.

[goBack: \(\) => void](#)

The `goBack` function allows navigating to the previous path.

[location: NavigatorLocation](#)

The `location` object represent the current location, and has a few properties:

- `path: string`. The path associated to the location.
- `isBack: boolean`. A flag that is `true` when the current location was reached by navigating backwards in the location stack.
- `isInitial: boolean`. A flag that is `true` only for the first (root) location in the location stack.

[params: Record< string, string | string\[\] >](#)

The parsed record of parameters from the current location. For example if the current screen path is `/product/:productId` and the location is `/product/123`, then `params` will be `{ productId: '123' }`.

First published

September 28, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NavigatorProvider](#)

[Previous NavigatorButton](#) [Previous: NavigatorButton](#)

[Next NavigatorScreen](#) [Next: NavigatorScreen](#)

NavigatorScreen

In this article

Table of Contents

- [Usage](#)

- [Props](#)
 - [path: string](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

The `NavigatorScreen` component represents a single view/screen/panel and should be used in combination with the [NavigatorProvider](#), the [NavigatorButton](#) and the [NavigatorBackButton](#) components (or the `useNavigator` hook).

Usage

Refer to [the NavigatorProvider component](#) for a usage example.

Props

The component accepts the following props:

[path: string](#)

The screen’s path, matched against the current path stored in the navigator.

- Required: Yes

First published

September 28, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NavigatorScreen”](#)

[Previous NavigatorProvider](#) [Previous: NavigatorProvider](#)

[Next NavigatorToParentButton](#) [Next: NavigatorToParentButton](#)

NavigatorToParentButton

In this article

Table of Contents

- [Usage](#)
 - [Inherited props](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

The `NavigatorToParentButton` component can be used to navigate to a screen and should be used in combination with the [NavigatorProvider](#), the [NavigatorScreen](#) and the [NavigatorButton](#) components (or the `useNavigator` hook).

Usage

Refer to [the NavigatorProvider component](#) for a usage example.

Inherited props

`NavigatorToParentButton` also inherits all of the [Button props](#), except for `href` and `target`.

First published

February 13, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NavigatorToParentButton”](#)

[Previous NavigatorScreen](#) [Previous: NavigatorScreen](#)

[Next Notice](#) [Next: Notice](#)

Notice

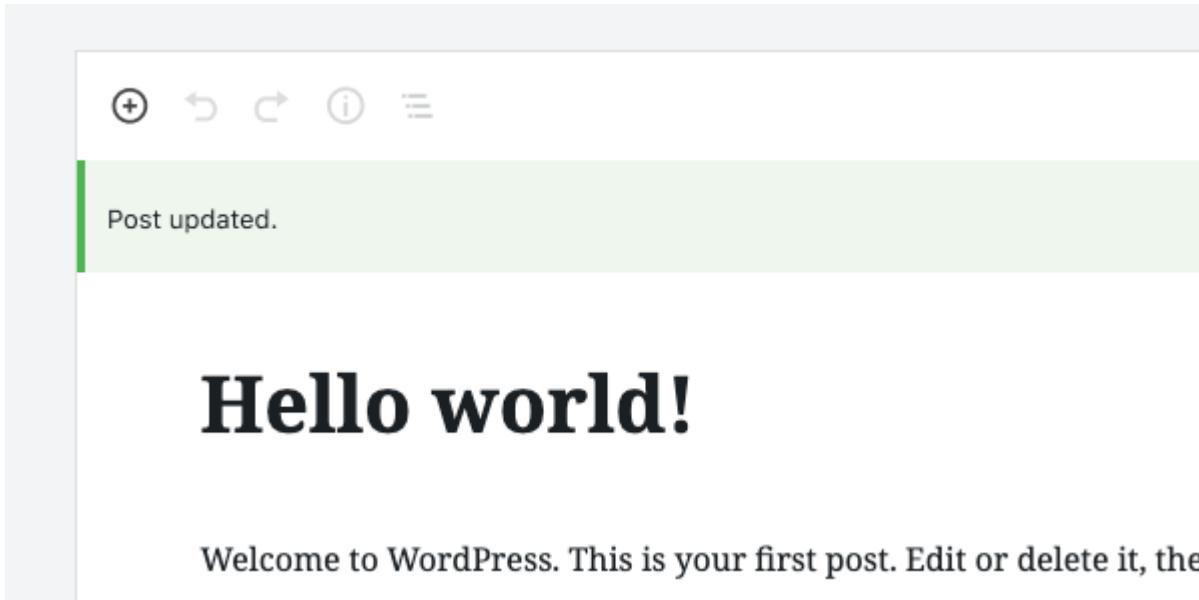
In this article

Table of Contents

- [Design guidelines](#)
 - [Usage](#)
 - [Do’s and Don’ts](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

Use Notices to communicate prominent messages to the user.



Design guidelines

A Notice displays a succinct message. It can also offer the user options, like viewing a published post or updating a setting, and requires a user action to be dismissed.

Use Notices to communicate things that are important but don't necessarily require action — a user can keep using the product even if they don't choose to act on a Notice. They are less interruptive than a Modal.

Usage

Notices display at the top of the screen, below any toolbars anchored to the top of the page. They're persistent and non-modal. Since they don't overlay the content, users can ignore or dismiss them, and choose when to interact with them.

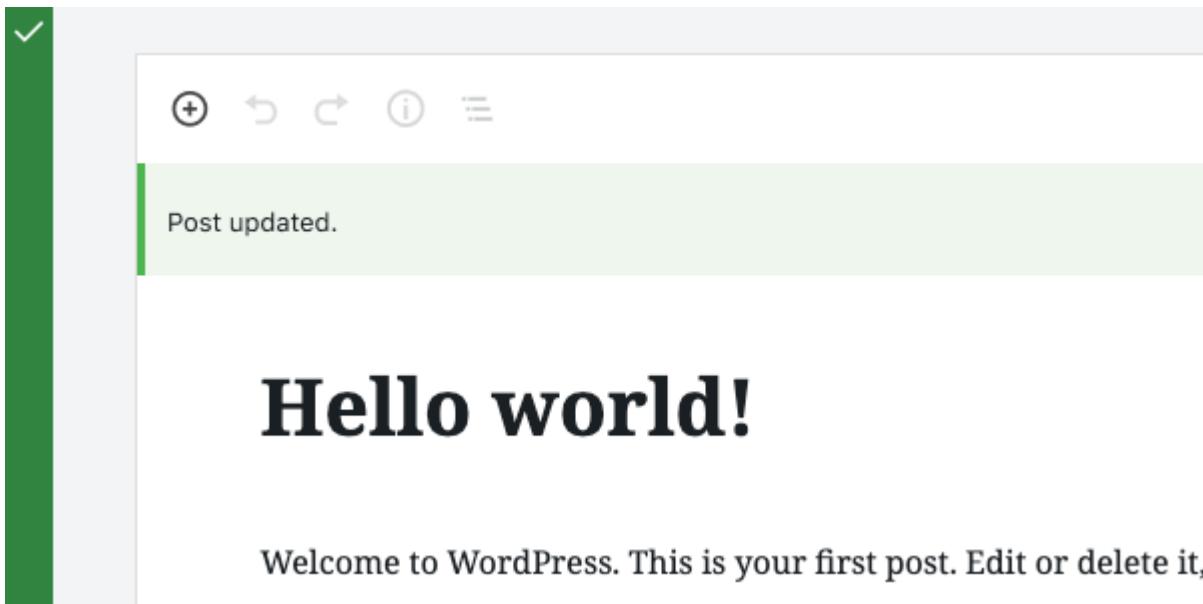
Notices are color-coded to indicate the type of message being communicated:

- **Informational** notices are **blue** by default.
- If there is a parent Theme component with an `accent` color prop, informational notices will take on that color instead.
- **Success** notices are **green**.
- **Warning** notices are **yellow****.**
- **Error** notices are **red**.

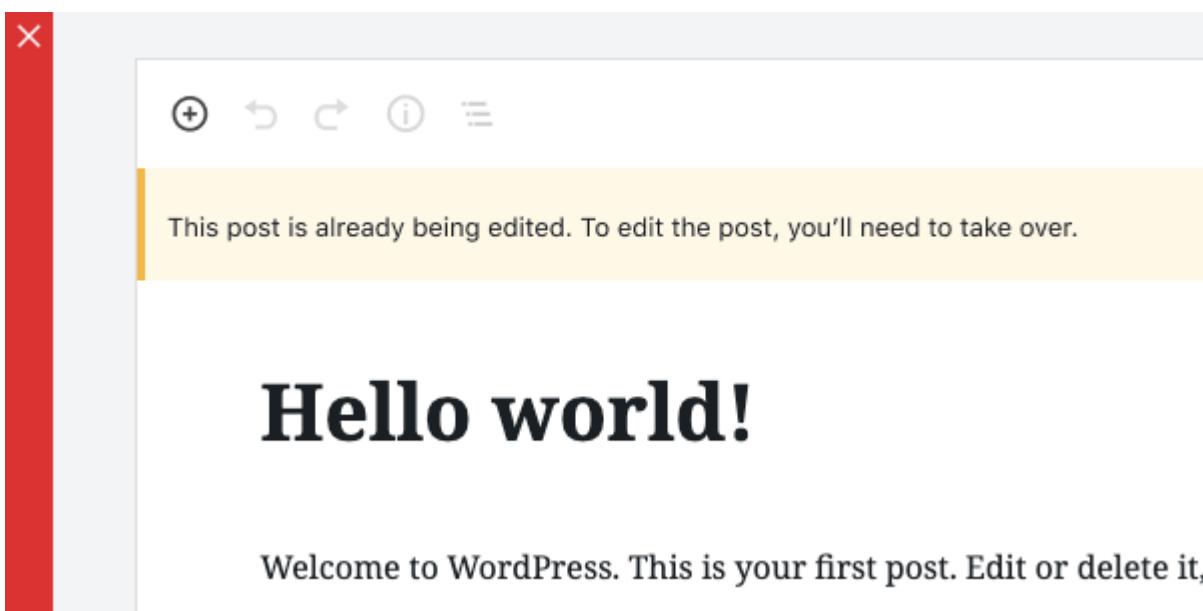
If an icon is included in the Notice, it should be color-coded to match the Notice state.

Do's and Don'ts

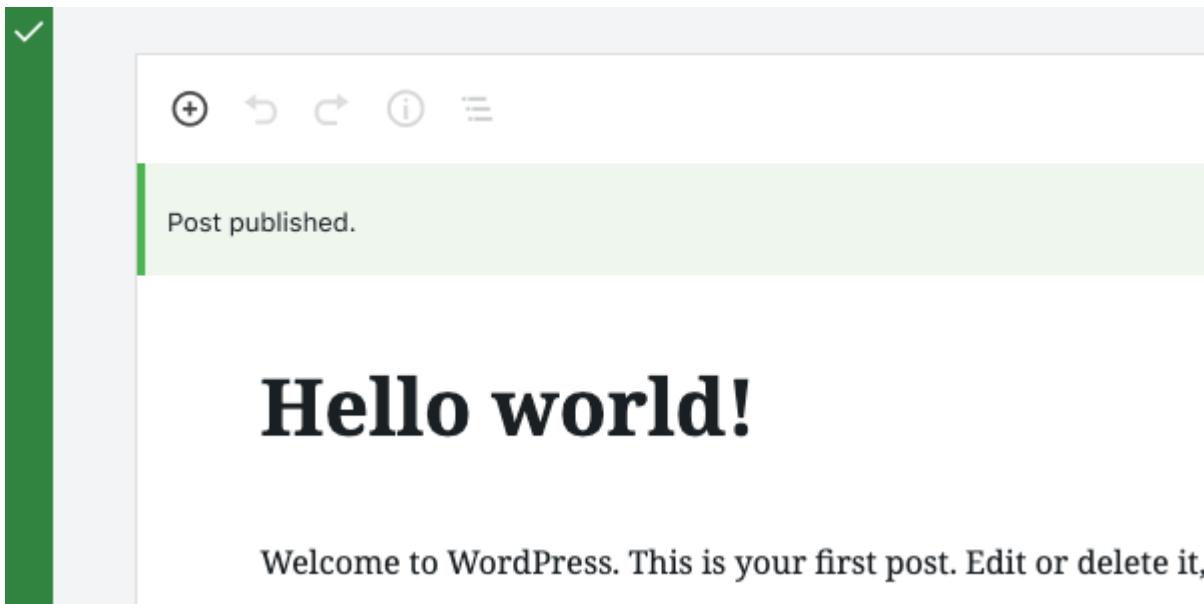
Do use a Notice when you want to communicate a message of medium importance.



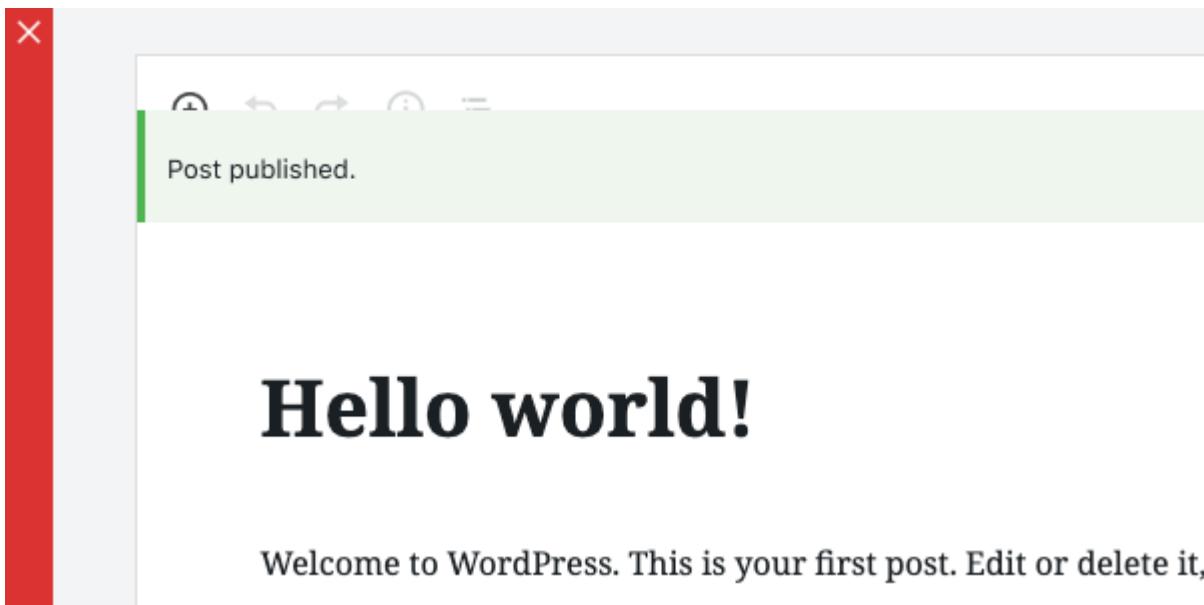
Don't use a Notice for a message that requires immediate attention and action from the user. Use a Modal for this instead.



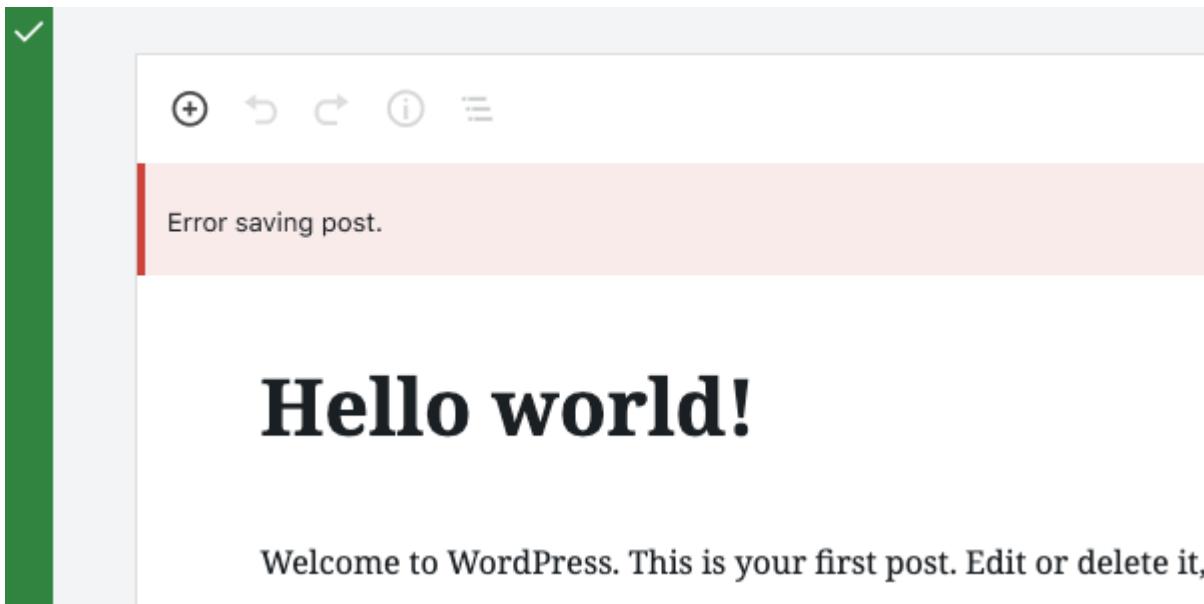
Do display Notices at the top of the screen, below any toolbars.



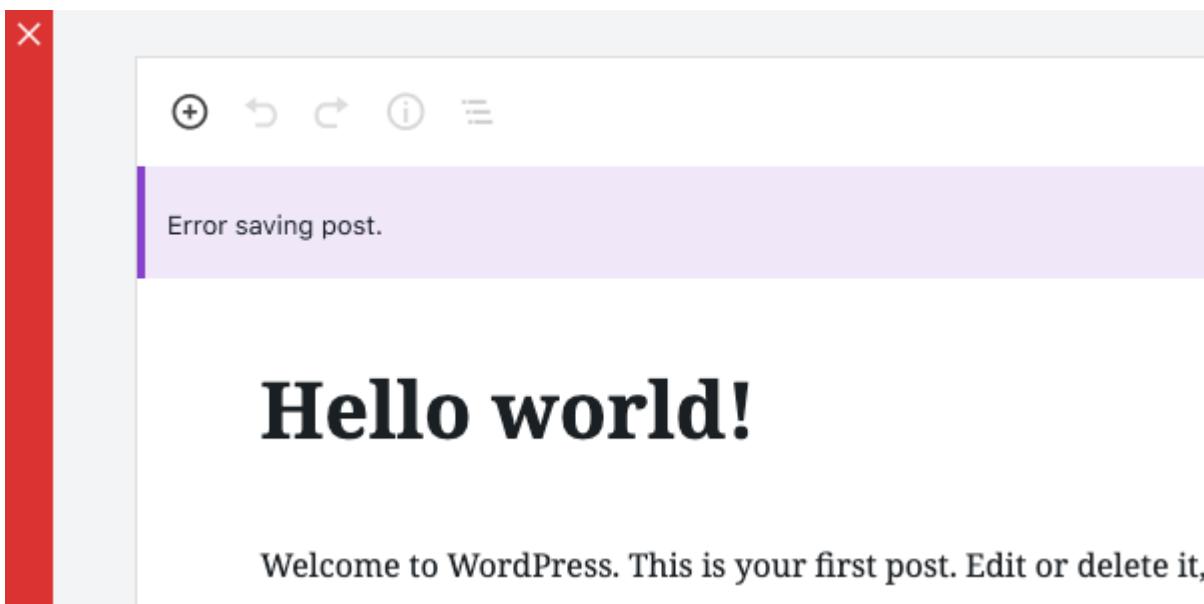
Don't show Notices on top of toolbars.



Do use color to indicate the type of message being communicated.



Don't apply any colors other than those for Warnings, Success, or Errors.



Development guidelines

Usage

To display a plain notice, pass `Notice` a string:

```
import { Notice } from `@wordpress/components`;  
  
const MyNotice = () => (  
    <Notice status="error">An unknown error occurred.</Notice>  
) ;
```

For more complex markup, you can pass any JSX element:

```
import { Notice } from `@wordpress/components`;

const MyNotice = () => (
    <Notice status="error">
        <p>
            An error occurred: <code>{ errorDetails }</code>.
        </p>
    </Notice>
);


```

Props

The following props are used to control the behavior of the component.

children: ReactNode

The displayed message of a notice. Also used as the spoken message for assistive technology, unless `spokenMessage` is provided as an alternative message.

- Required: Yes

spokenMessage: ReactNode

Used to provide a custom spoken message in place of the `children` default.

- Required: No
- Default: `children`

status: 'warning' | 'success' | 'error' | 'info'

Determines the color of the notice: `warning` (yellow), `success` (green), `error` (red), or '`info`'. By default '`info`' will be blue, but if there is a parent Theme component with an accent color prop, the notice will take on that color instead.

- Required: No
- Default: `info`

onRemove: () => void

A function called to dismiss/remove the notice.

- Required: No
- Default: `noop`

politeness: 'polite' | 'assertive'

A politeness level for the notice's spoken message. Should be provided as one of the valid options for [an aria-live attribute value](#).

- A value of '`assertive`' is to be used for important, and usually time-sensitive, information. It will interrupt anything else the screen reader is announcing in that moment.

- A value of 'polite' is to be used for advisory information. It should not interrupt what the screen reader is announcing in that moment (the “speech queue”) or interrupt the current task.

Note that this value should be considered a suggestion; assistive technologies may override it based on internal heuristics.

- Required: No
- Default: 'assertive' or 'polite', based on the notice status.

`isDismissible: boolean`

Whether the notice should be dismissible or not

- Required: No
- Default: true

`onDismiss: () => void`

A deprecated alternative to `onRemove`. This prop is kept for compatibility reasons but should be avoided.

- Required: No
- Default: noop

`actions: Array<NoticeAction>.`

An array of notice actions. Each member object should contain:

- `label: string` containing the text of the button/link
- `url: string` OR `onClick: (event: SyntheticEvent) => void` to specify what the action does.
- `className: string` (optional) to add custom classes to the button styles.
- `noDefaultClasses: boolean` (optional) A value of true will remove all default styling.
- `variant: 'primary' | 'secondary' | 'link'` (optional) You can denote a primary button action for a notice by passing a value of `primary`.

The default appearance of an action button is inferred based on whether `url` or `onClick` are provided, rendering the button as a link if appropriate. If both props are provided, `url` takes precedence, and the action button will render as an anchor tag.

Related components

- To create a more prominent message that requires action, use a Modal.
- For low priority, non-interruptive messages, use Snackbar.

First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: Notice”](#)

[Previous NavigatorToParentButton](#) [Previous: NavigatorToParentButton](#)

[Next Navigator](#) [Next: Navigator](#)

Navigator

In this article

[Table of Contents](#)

- [Usage](#)
- [Navigator Props](#)
 - [initialPath](#)
- [NavigatorScreen Props](#)
 - [path](#)
- [The navigator object.](#)
 - [push](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

The Navigator components allows rendering nested panels or menus (also called screens) and navigate between these different states. The Global Styles sidebar is an example of this.

The components is not opinionated in terms of UI, it lets compose any UI components to navigate between the nested screens.

[Usage](#)

```
import {  
    __experimentalNavigator as Navigator,  
    __experimentalNavigatorScreen as NavigatorScreen,  
    __experimentalUseNavigator as useNavigator,  
} from '@wordpress/components';  
  
function NavigatorButton( {  
    path,  
    isBack = false,  
    ...props  
} ) {  
    const navigator = useNavigator();  
    return (  
        <Button
```

```

        onClick={ () => navigator.push( path, { isBack } ) }
        {...props}
      />
    );
}

const MyNavigation = () => (
  <Navigator initialPath="/">
    <NavigatorScreen path="/">
      <p>This is the home screen.</p>
      <NavigatorButton isPrimary path="/child">
        Navigate to child screen.
      </NavigatorButton>
    </NavigatorScreen>

    <NavigatorScreen path="/child">
      <p>This is the child screen.</p>
      <NavigatorButton isPrimary path="/" isBack>
        Go back
      </NavigatorButton>
    </NavigatorScreen>
  </Navigator>
);

```

Navigator Props

Navigator supports the following props.

initialPath

- Type: `string`
- Required: No

The initial active path.

NavigatorScreen Props

NavigatorScreen supports the following props.

path

- Type: `string`
- Required: Yes

The path of the current screen.

The navigator object.

You can retrieve a `navigator` instance by using the `useNavigator` hook.
The navigator offers the following methods:

[push](#)

- Type: (path: string, options) => void

The `push` function allows you to navigate to a given path. The second argument can augment the navigation operations with different options.

The available options are:

- `isBack` (boolean): A boolean flag indicating that we're moving back to a previous state.

First published

September 23, 2021

Last updated

September 28, 2021

Edit article

[Improve it on GitHub: Navigator”](#)

[Previous Notice](#) [Previous: Notice](#)
[Next NumberControl](#) [Next: NumberControl](#)

NumberControl

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [dragDirection](#)
 - [dragThreshold](#)
 - [spinControls](#)
 - [isDragEnabled](#)
 - [isShiftStepEnabled](#)
 - [label](#)
 - [labelPosition](#)
 - [max](#)
 - [min](#)
 - [onChange](#)
 - [required](#)
 - [shiftStep](#)
 - [step](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

NumberControl is an enhanced HTML `input[type="number"]` element.

Usage

```
import { __experimentalNumberControl as NumberControl } from '@wordpress/c

const Example = () => {
  const [ value, setValue ] = useState( 10 );

  return (
    <NumberControl
      isShiftStepEnabled={ true }
      onChange={ setValue }
      shiftStep={ 10 }
      value={ value }
    />
  );
};
```

Props

dragDirection

Determines the drag axis to increment/decrement the value.

Directions: n | e | s | w

- Type: String
- Required: No
- Default: n

dragThreshold

If `isDragEnabled` is true, this controls the amount of px to have been dragged before the value changes.

- Type: Number
- Required: No
- Default: 10

spinControls

The type of spin controls to display. These are buttons that allow the user to quickly increment and decrement the number.

- ‘none’ – Do not show spin controls.
- ‘native’ – Use browser’s native HTML `input` controls.
- ‘custom’ – Use plus and minus icon buttons.
 - Type: String
 - Required: No
 - Default: ‘native’

isDragEnabled

If true, enables mouse drag gesture to increment/decrement the number value. Holding SHIFT while dragging will increase the value by the shiftStep.

- Type: Boolean
- Required: No

isShiftStepEnabled

If true, pressing UP or DOWN along with the SHIFT key will increment the value by the shiftStep value.

- Type: Boolean
- Required: No
- Default: true

label

If this property is added, a label will be generated using label property as the content.

- Type: String
- Required: No

labelPosition

The position of the label (top, side, bottom, or edge).

- Type: String
- Required: No

max

The maximum value allowed.

- Type: Number
- Required: No
- Default: Infinity

min

The minimum value allowed.

- Type: Number
- Required: No
- Default: -Infinity

onChange

Callback fired whenever the value of the input changes.

The callback receives two arguments:

1. `newValue`: the new value of the input
2. `extra`: an object containing, under the `event` key, the original browser event.

Note that the value received as the first argument of the callback is *not* guaranteed to be a valid value (e.g. it could be outside of the range defined by the `[min, max]` props, or it could not match the `step`). In order to check the value's validity, check the `event.target?.validity.valid` property from the callback's second argument.

- Type: `(newValue, extra) => void`
- Required: No

required

If `true` enforces a valid number within the control's min/max range. If `false` allows an empty string as a valid value.

- Type: Boolean
- Required: No
- Default: `false`

shiftStep

Amount to increment by when the SHIFT key is held down. This shift value is a multiplier to the `step` value. For example, if the `step` value is 5, and `shiftStep` is 10, each jump would increment/decrement by 50.

- Type: Number
- Required: No
- Default: 10

step

Amount by which the `value` is changed when incrementing/decrementing. It is also a factor in validation as `value` must be a multiple of `step` (offset by `min`, if specified) to be valid.

Accepts the special string value `any` that voids the validation constraint and causes stepping actions to increment/decrement by 1.

- Type: Number | "any"
- Required: No
- Default: 1

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: NumberControl](#)”

[Previous](#) [Navigator](#) [Previous: Navigator](#)[Next Panel](#) [Next: Panel](#)

Panel

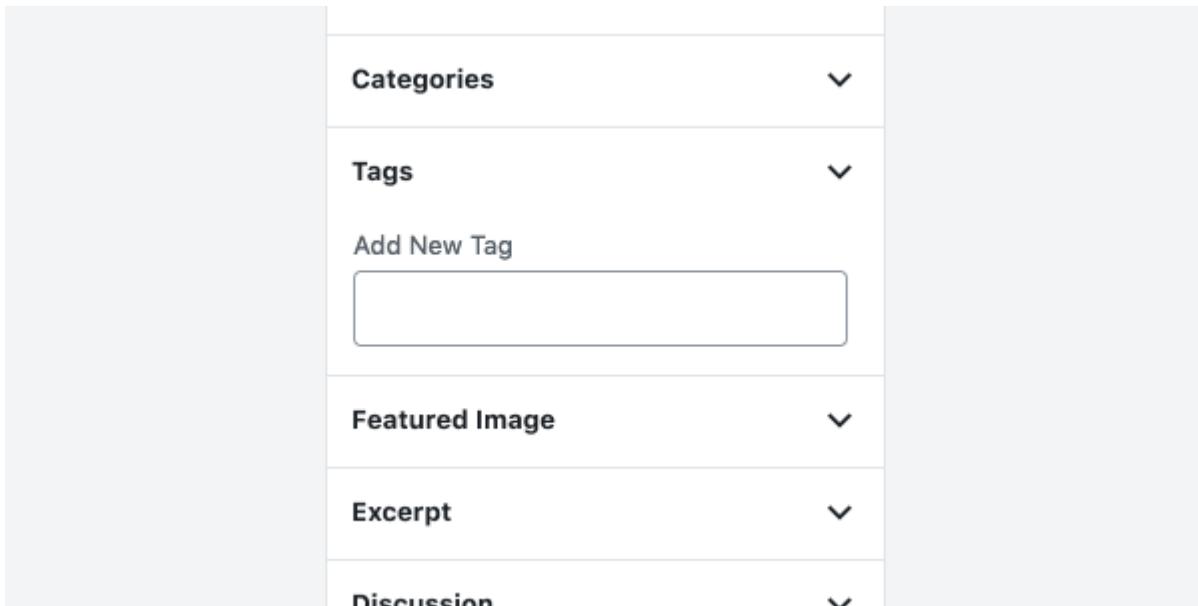
In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Anatomy](#)
 - [Usage](#)
 - [Behavior](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Sub-Components](#)
- [Related components](#)

[↑ Back to top](#)

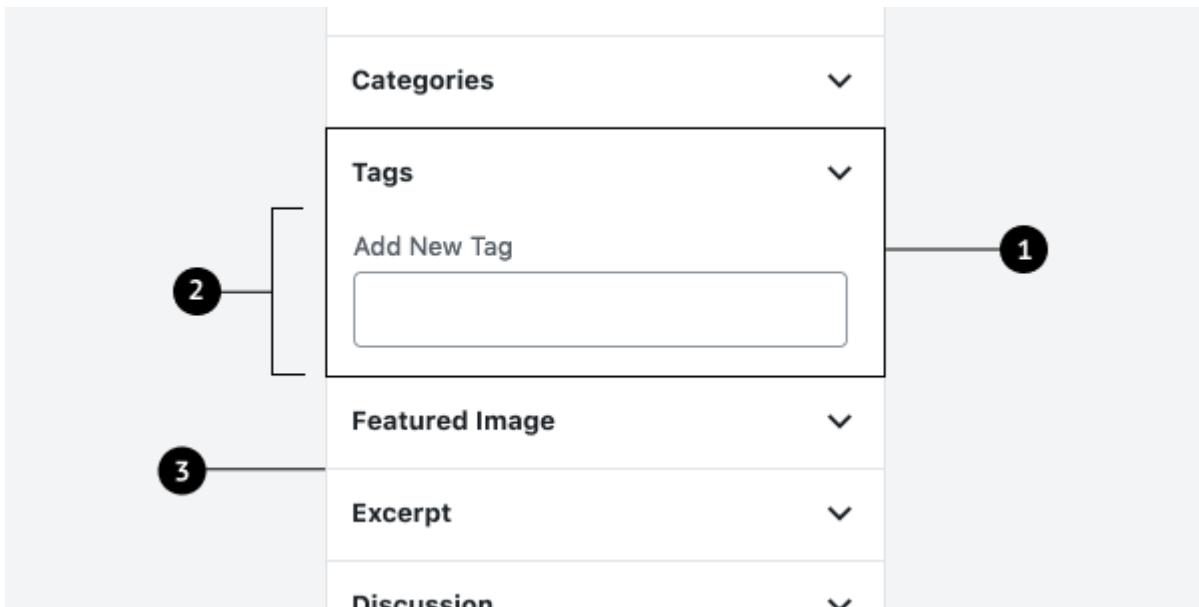
Panels expand and collapse multiple sections of content.



[Design guidelines](#)

[Anatomy](#)

A Panel is a single section of content that can be expanded or collapsed as needed.



1. Panel
2. Body
3. Divider

Usage

Panels show and hide details of list items by expanding and collapsing list content vertically. Panels help users see only the content they need.

When to use Panels

Use Panels when it's helpful to:

- See an overview of multiple, related sections of content.
- Show and hide those sections as needed.
- Hide information that is lower priority that users don't need to see all the time.
- View more than one section at a time.

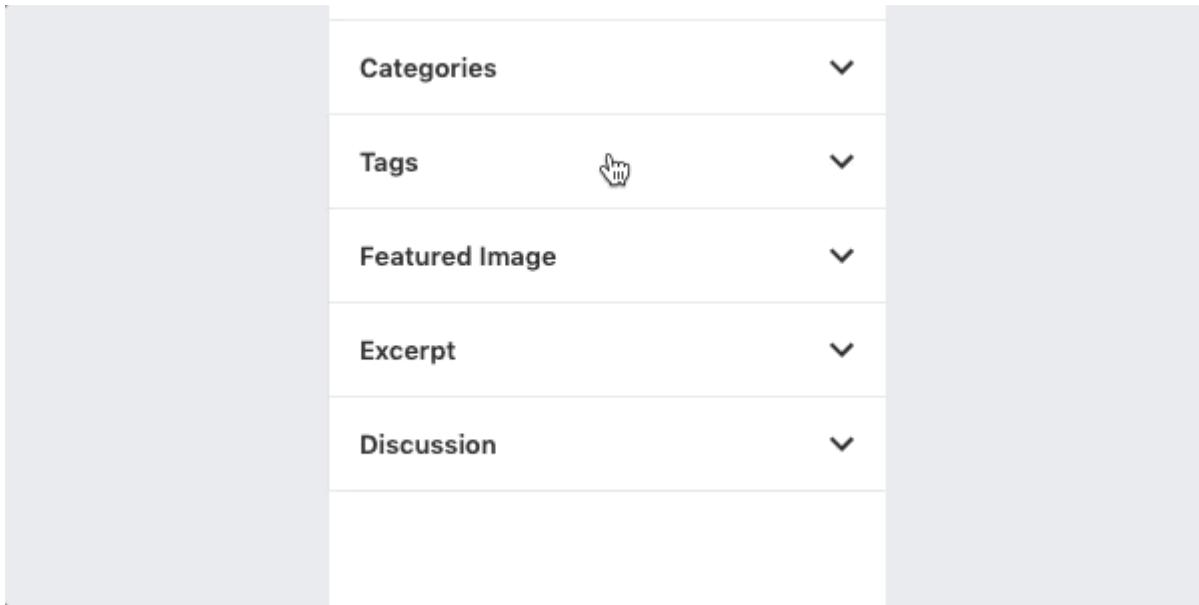
Consider an alternative component when:

- There's crucial information or error messages that require immediate action.
- You need to quickly switch between only a few sections (consider using Tabs instead).

Behavior

Expand and collapse

Show and hide details of existing panel items by expanding and collapsing list content vertically.



Collapsible panels are indicated with a caret icon that is flipped when expanded. Carets are preferable over a plus or arrow icon, because a plus indicates addition, and arrows are closely related to navigation.

Panels should be expanded by default if the content is important or essential. Panels that are open by default should appear at the top.

Development guidelines

The `Panel` creates a container with a header that can take collapsible `PanelBody` components to easily create a user friendly interface for affecting state and attributes.

Usage

```
import { Panel, PanelBody, PanelRow } from '@wordpress/components';
import { more } from '@wordpress/icons';

const MyPanel = () => (
  <Panel header="My Panel">
    <PanelBody title="My Block Settings" icon={ more } initialOpen={ true }>
      <PanelRow>My Panel Inputs and Labels</PanelRow>
    </PanelBody>
  </Panel>
);
```

Sub-Components

Panel

Props

`header: string`

The text that will be rendered as the title of the panel. Text will be rendered inside an `<h2>` tag.

- Required: No

`className: string`

The CSS class to apply to the wrapper element.

- Required: No

`children: React.ReactNode`

The content to display within the panel row.

- Required: Yes
-

PanelBody

The PanelBody creates a collapsible container that can be toggled open or closed.

Props

`title: string`

Title text. It shows even when the component is closed.

- Required: No

`opened: boolean`

When set to `true`, the component will remain open regardless of the `initialOpen` prop and the panel will be prevented from being closed.

- Required: No

`className: string`

The CSS class to apply to the wrapper element.

- Required: No

```
icon: JSX.Element
```

An icon to be shown next to the title.

- Required: No

```
onToggle: ( next: boolean ) => void;
```

A function that is called any time the component is toggled from its closed state to its opened state, or vice versa.

- Required: No
- Default: `noop`

```
initialOpen: boolean
```

Whether or not the panel will start open.

- Required: No
- Default: `true`

```
children: | React.ReactNode | ( ( props: { opened: boolean } ) => React.ReactNode )
```

The content to display in the **PanelBody**. If a function is provided for this prop, it will receive an object with the `opened` prop as an argument.

- Required: No

```
buttonProps: WordPressComponentProps<Omit< ButtonAsButtonProps, 'icon' >, 'button', false>
```

Props that are passed to the **Button** component in title within the **PanelBody**.

- Required: No
- Default: `{}`

```
scrollAfterOpen: boolean
```

Scrolls the content into view when visible. This improves the UX when multiple **PanelBody** components are stacked in a scrollable container.

- Required: No
- Default: `true`

PanelRow

PanelRow is a generic container for rows within a **PanelBody**. It is a flex container with a top margin for spacing.

Props

`className: string`

The CSS class to apply to the wrapper element.

- Required: No

`children: React.ReactNode`

The content to display within the panel row.

- Required: No

Ref

`PanelRow` accepts a forwarded ref that will be added to the wrapper div. Usage:

```
<PanelRow className="edit-post-post-schedule" ref={ panelRowRef }>
```

PanelHeader

`PanelHeader` renders the header for the `Panel`. This is used by the `Panel` component under the hood, so it does not typically need to be used.

Props

`label: string`

The text that will be rendered as the title of the `Panel`. Will be rendered in an `<h2>` tag.

- Required: No

`children: React.ReactNode`

The content to display within the panel row.

- Required: No

Related components

- To divide related sections of content accessed by a horizontal menu, use `TabPanel`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Panel”](#)

[Previous NumberControl](#) [Previous: NumberControl](#)

[Next Placeholder](#) [Next: Placeholder](#)

Placeholder

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [className: string](#)
 - [icon: string|Function|Component|null](#)
 - [instructions: string](#)
 - [isColumnLayout: boolean](#)
 - [label: string](#)
 - [notices: ReactNode](#)
 - [preview: ReactNode](#)
 - [withIllustration: boolean](#)

[↑ Back to top](#)

Usage

```
import { Placeholder } from '@wordpress/components';
import { more } from '@wordpress/icons';

const MyPlaceholder = () => <Placeholder icon={ more } label="Placeholder"
```

Props

[className: string](#)

Class to set on the container div.

- Required: No

[icon: string|Function|Component|null](#)

If provided, renders an icon next to the label.

- Required: No

[instructions: string](#)

Instructions of the placeholder.

- Required: No

[isColumnLayout: boolean](#)

Changes placeholder children layout from flex-row to flex-column.

- Required: No

[label: string](#)

Title of the placeholder.

- Required: No

[notices: ReactNode](#)

A rendered notices list

- Required: No

[preview: ReactNode](#)

Preview to be rendered in the placeholder.

- Required: No

[withIllustration: boolean](#)

Outputs a placeholder illustration.

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Placeholder”](#)

[Previous Panel](#) [Previous: Panel](#)
[Next Popover](#) [Next: Popover](#)

Popover

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [anchor: Element | VirtualElement | null](#)
 - [anchorRect: DomRectWithOwnerDocument](#)
 - [anchorRef: Element | PopoverAnchorRefReference | PopoverAnchorRefTopBottom | Range](#)
 - [animate: boolean](#)
 - [children: ReactNode](#)
 - [expandOnMobile: boolean](#)
 - [flip: boolean](#)
 - [focusOnMount: 'firstElement' | boolean](#)
 - [onFocusOutside: \(event: SyntheticEvent \) => void](#)
 - [getAnchorRect: \(fallbackReferenceElement: Element | null \) => DomRectWithOwnerDocument](#)
 - [headerTitle: string](#)
 - [isAlternate: boolean](#)
 - [noArrow: boolean](#)
 - [offset: number](#)
 - [onClose: \(\) => void](#)
 - [placement: 'top' | 'top-start' | 'top-end' | 'right' | 'right-start' | 'right-end' | 'bottom' | 'bottom-start' | 'bottom-end' | 'left' | 'left-start' | 'left-end' | 'overlay'](#)
 - [position: \[yAxis\] \[xAxis\] \[optionalCorner\]](#)
 - [resize: boolean](#)
 - [variant: 'toolbar' | 'unstyled'](#)

[↑ Back to top](#)

Popover renders its content in a floating modal. If no explicit anchor is passed via props, it anchors to its parent element by default.

The behavior of the popover when it exceeds the viewport's edges can be controlled via its props.

Usage

Render a Popover adjacent to its container.

If a Popover is returned by your component, it will be shown. To hide the popover, simply omit it from your component's render value.

```
import { useState } from 'react';
import { Button, Popover } from '@wordpress/components';

const MyPopover = () => {
  const [ isVisible, setIsVisible ] = useState( false );
  const toggleVisible = () => {
```

```

        setIsVisible( ( state ) => ! state );
    };

    return (
        <Button variant="secondary" onClick={ toggleVisible }>
            Toggle Popover!
            { isVisible && <Popover>Popover is toggled!</Popover> }
        </Button>
    );
};


```

In order to pass an explicit anchor, you can use the `anchor` prop. When doing so, **the anchor element should be stored in local state** rather than a plain React ref to ensure reactive updating when it changes.

```

import { useState } from 'react';
import { Button, Popover } from '@wordpress/components';

const MyPopover = () => {
    // Use internal state instead of a ref to make sure that the component
    // re-renders when the popover's anchor updates.
    const [ popoverAnchor, setPopoverAnchor ] = useState();
    const [ isVisible, setIsVisible ] = useState( false );
    const toggleVisible = () => {
        setIsVisible( ( state ) => ! state );
    };

    return (
        <p ref={ setPopoverAnchor }>Popover s anchor</p>
        <Button variant="secondary" onClick={ toggleVisible }>
            Toggle Popover!
        </Button>
        { isVisible && (
            <Popover
                anchor={ popoverAnchor }
            >
                Popover is toggled!
            </Popover>
        ) }
    );
};


```

By default Popovers render at the end of the body of your document. If you want Popover elements to render to a specific location on the page, you must render a `Popover.Slot` further up the element tree:

```

import { createRoot } from 'react-dom/client';
import { Popover } from '@wordpress/components';
import Content from './Content';

const app = document.getElementById( 'app' );
const root = createRoot( app );
root.render(
    <div>

```

```
<Content />
<Popover.Slot />
</div>
);
```

Props

The component accepts the following props. Props not included in this set will be applied to the element wrapping Popover content.

anchor: Element | VirtualElement | null

The element that should be used by the Popover as its anchor. It can either be an Element or, alternatively, a VirtualElement — ie. an object with the getBoundingClientRect() and the ownerDocument properties defined.

The element should be stored in state rather than a plain ref to ensure reactive updating when it changes.

- Required: No

anchorRect: DomRectWithOwnerDocument

Note: this prop is deprecated. Please use the anchor prop instead.

An object extending a DOMRect with an additional optional ownerDocument property, used to specify a fixed popover position.

- Required: No

anchorRef: Element | PopoverAnchorRefReference | PopoverAnchorRefTopBottom | Range

Note: this prop is deprecated. Please use the anchor prop instead.

Used to specify a fixed popover position. It can be an Element, a React reference to an element, an object with a top and a bottom properties (both pointing to elements), or a range.

- Required: No

animate: boolean

Whether the popover should animate when opening.

- Required: No
- Default: true

children: ReactNode

The children elements rendered as the popover's content.

- Required: Yes

[expandOnMobile: boolean](#)

Show the popover fullscreen on mobile viewports.

- Required: No

[flip: boolean](#)

Specifies whether the popover should flip across its axis if there isn't space for it in the normal placement.

When using a 'top' placement, the popover will switch to a 'bottom' placement. When using a 'left' placement, the popover will switch to a 'right' placement.

The popover will retain its alignment of 'start' or 'end' when flipping.

- Required: No
- Default: true

[focusOnMount: 'firstElement' | boolean](#)

By default, the *first tabbable element* in the popover will receive focus when it mounts. This is the same as setting this prop to "firstElement".

Specifying a true value will focus the container instead.

Specifying a false value disables the focus handling entirely (this should only be done when an appropriately accessible substitute behavior exists).

- Required: No
- Default: "firstElement"

[onFocusOutside: \(event: SyntheticEvent \)=> void](#)

A callback invoked when the focus leaves the opened popover. This should only be provided in advanced use-cases when a popover should close under specific circumstances (for example, if the new document.activeElement is content of or otherwise controlling popover visibility).

When not provided, the onClose callback will be called instead.

- Required: No

[getAnchorRect: \(fallbackReferenceElement: Element | null \)=> DomRectWithOwnerDocument](#)

Note: this prop is deprecated. Please use the anchor prop instead.

A function returning the same value as the one expected by the anchorRect prop, used to specify a dynamic popover position.

- Required: No

[headerTitle: string](#)

Used to customize the header text shown when the popover is toggled to fullscreen on mobile viewports (see the `expandOnMobile` prop).

- Required: No

[isAlternate: boolean](#)

Note: this prop is deprecated. Please use the `variant` prop with the 'toolbar' values instead.

Used to enable a different visual style for the popover.

- Required: No

[noArrow: boolean](#)

Used to show/hide the arrow that points at the popover's anchor.

- Required: No
- Default: true

[offset: number](#)

The distance (in px) between the anchor and the popover.

- Required: No

[onClose: \(\) => void](#)

A callback invoked when the popover should be closed.

- Required: No

[placement: 'top' | 'top-start' | 'top-end' | 'right' | 'right-start' | 'right-end' | 'bottom' | 'bottom-start' | 'bottom-end' | 'left' | 'left-start' | 'left-end' | 'overlay'](#)

Used to specify the popover's position with respect to its anchor.

`overlay` is a special case that places the popover over the reference element.
Please note that other placement related props may not behave as expected.

- Required: No
- Default: "bottom-start"

[position: \[yAxis\] \[xAxis\] \[optionalCorner\]](#)

Note: use the `placement` prop instead when possible.

Legacy way to specify the popover's position with respect to its anchor.

Possible values:

- `yAxis: 'top' | 'middle' | 'bottom'`
- `xAxis: 'left' | 'center' | 'right'`
- `corner: 'top' | 'right' | 'bottom' | 'left'`
- Required: No

[resize: boolean](#)

Adjusts the size of the popover to prevent its contents from going out of view when meeting the viewport edges.

- Required: No
- Default: `true`

[variant: ‘toolbar’ | ‘unstyled’](#)

Specifies the popover's style.

Leave undefined for the default style. Possible values are:

- `unstyled`: The popover is essentially without any visible style, it has no background, border, outline or drop shadow, but the popover contents are still displayed.
- `toolbar`: A style that has no elevation, but a high contrast with other elements. This matches the style of the [Toolbar component](#).

– Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Popover”](#)

[Previous Placeholder](#) [Placeholder Previous: Placeholder](#)
[Next QueryControls](#) [QueryControls Next: QueryControls](#)

QueryControls

In this article

Table of Contents

- [Development Guidelines](#)
 - [Usage](#)
 - [Multiple category selector](#)
 - [Props](#)

[↑ Back to top](#)

Development Guidelines

Usage

```
import { useState } from 'react';
import { QueryControls } from '@wordpress/components';

const QUERY_DEFAULTS = {
    category: 1,
    categories: [
        {
            id: 1,
            name: 'Category 1',
            parent: 0,
        },
        {
            id: 2,
            name: 'Category 1b',
            parent: 1,
        },
        {
            id: 3,
            name: 'Category 2',
            parent: 0,
        },
    ],
    maxItems: 20,
    minItems: 1,
    numberofItems: 10,
    order: 'asc',
    orderBy: 'title',
};

const MyQueryControls = () => {
    const [ query, setQuery ] = useState( QUERY_DEFAULTS );
    const { category, categories, maxItems, minItems, numberofItems, order }

    const updateQuery = ( newQuery ) => {
        setQuery( { ...query, ...newQuery } );
    };

    return (
        <QueryControls
            { ...{ maxItems, minItems, numberofItems, order, orderBy } }
            onOrderByChange={ ( newOrderBy ) => updateQuery( { orderBy: newOrderBy } ) }
            onOrderChange={ ( newOrder ) => updateQuery( { order: newOrder } ) }
            categoriesList={ categories }
            selectedCategoryId={ category }
            onCategoryChange={ ( newCategory ) => updateQuery( { category: newCategory } ) }
            onNumberOfItemsChange={ ( newNumberOfItems ) => updateQuery( { numberofItems: newNumberOfItems } ) }
        >
    );
}
```

```

        updateQuery( { numberOfItems: newNumberOfItems } )
    }
  />
);
};

```

Multiple category selector

The `QueryControls` component now supports multiple category selection, to replace the single category selection available so far. To enable it use the component with the new props instead: `categorySuggestions` in place of `categoriesList` and the `selectedCategories` array instead of `selectedCategoryId` like so:

```

const QUERY_DEFAULTS = {
  orderBy: 'title',
  order: 'asc',
  selectedCategories: [
    {
      id: 1,
      value: 'Category 1',
      parent: 0,
    },
    {
      id: 2,
      value: 'Category 1b',
      parent: 1,
    },
  ],
  categories: {
    'Category 1': {
      id: 1,
      name: 'Category 1',
      parent: 0,
    },
    'Category 1b': {
      id: 2,
      name: 'Category 1b',
      parent: 1,
    },
    'Category 2': {
      id: 3,
      name: 'Category 2',
      parent: 0,
    },
  },
  numberOfItems: 10,
};

const MyQueryControls = () => {
  const [ query, setQuery ] = useState( QUERY_DEFAULTS );
  const { orderBy, order, selectedCategories, categories, numberOfItems
  const updateQuery = ( newQuery ) => {

```

```

        setQuery( { ...query, ...newQuery } );
    };

    return (
        <QueryControls
            { ...{ orderBy, order, numberofItems } }
            onOrderByChange={ ( newOrderBy ) => updateQuery( { orderBy: newOrderBy } ) }
            onOrderChange={ ( newOrder ) => updateQuery( { order: newOrder } ) }
            categorySuggestions={ categories }
            selectedCategories={ selectedCategories }
            onCategoryChange={ ( category ) => updateQuery( { selectedCategories: [category] } ) }
            onNumberOfItemsChange={ ( newNumberOfItems ) =>
                updateQuery( { numberofItems: newNumberOfItems } )
            }
        />
    );
}
);

```

The format of the categories list also needs to be updated to match the expected type for the category suggestions.

Props

`authorList: Author[]`

An array of the authors to select from.

- Required: No
- Platform: Web

`categoriesList: Category[]`

An array of categories. When passed in conjunction with the `onCategoryChange` prop, it causes the component to render UI that allows selecting one category at a time.

- Required: No
- Platform: Web

`categorySuggestions: Record< Category['name'], Category >`

An object of categories with the category name as the key. When passed in conjunction with the `onCategoryChange` prop, it causes the component to render UI that enables multiple selection.

- Required: No
- Platform: Web

`maxItems: number`

The maximum number of items.

- Required: No
- Default: 100

- Platform: Web

`minItems: number`

The minimum number of items.

- Required: No
- Default: 1
- Platform: Web

`numberOfItems: number`

The selected number of items to retrieve via the query.

- Required: No
- Platform: Web

`onAuthorChange: (newAuthor: string) => void`

A function that receives the new author value. If not specified, the author controls are not rendered.

- Required: No
- Platform: Web

`onCategoryChange: (newCategory: string) => void | FormTokenFieldProps['onChange']`

A function that receives the new category value. If not specified, the category controls are not rendered.

The function's signature changes depending on whether multiple category selection is enabled or not.

- Required: No
- Platform: Web

`onNumberOfItemsChange: (newNumber?: number) => void`

A function that receives the new number of items. If not specified, then the number of items range control is not rendered.

- Required: No
- Platform: Web

`onOrderChange: (newOrder: 'asc' | 'desc') => void`

A function that receives the new order value. If this prop or the `onOrderByChange` prop are not specified, then the order controls are not rendered.

- Required: No
- Platform: Web

`onOrderByChange: (newOrderBy: 'date' | 'title') => void`

A function that receives the new orderby value. If this prop or the `onOrderChange` prop are not specified, then the order controls are not rendered.

- Required: No
- Platform: Web

`order: 'asc' | 'desc'`

The order in which to retrieve posts.

- Required: No
- Platform: Web

`orderBy: 'date' | 'title'`

The meta key by which to order posts.

- Required: No
- Platform: Web

`selectedAuthorId: number`

The selected author ID.

- Required: No
- Platform: Web

`selectedCategories: Category[]`

The selected categories for the `categorySuggestions` prop.

- Required: No
- Platform: Web

`selectedCategoryId: number`

The selected category for the `categoriesList` prop.

- Required: No
- Platform: Web

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: QueryControls”](#)

[Previous Popover](#) [Previous: Popover](#)
[Next RadioControl](#) [Next: RadioControl](#)

RadioControl

In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

Use radio buttons when you want users to select one option from a set, and you want to show them all the available options at once.



Selected and unselected radio buttons

Design guidelines

Usage

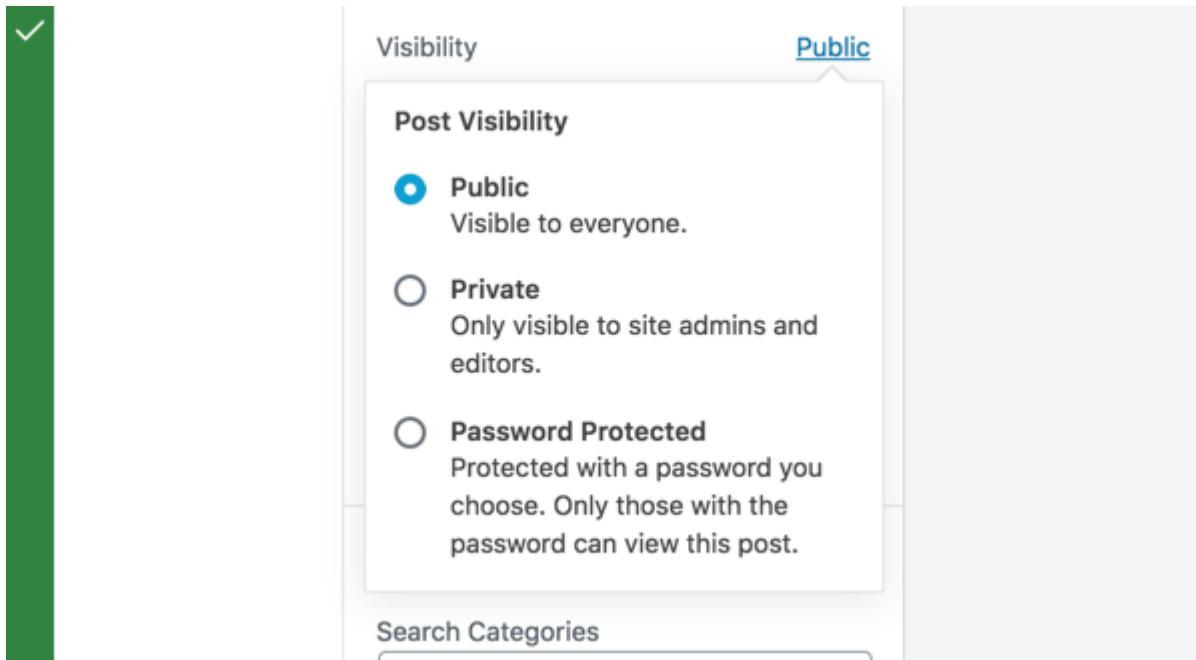
When to use radio buttons

Use radio buttons when you want users to:

- Select a single option from a list.
- Expose all available options.

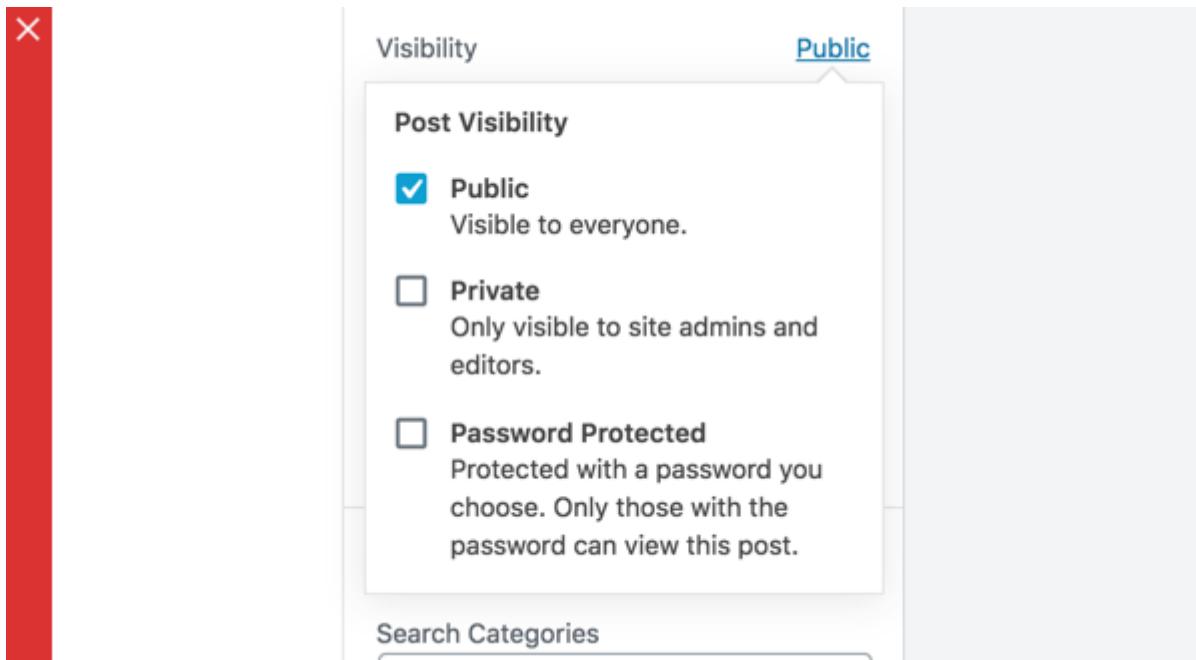
If you have a list of available options that can be collapsed, consider using a dropdown menu instead, as dropdowns use less space. A country selection field, for instance, would be very large as a group of radio buttons and wouldn't help the user gain more context by seeing all options at once.

Do



Use radio buttons when only one item can be selected from a list.

Don't



Don't use checkboxes when only one item can be selected from a list. Use radio buttons instead.

Defaults

When using radio buttons **one should be selected by default** (i.e., when the page loads, in the case of a web application).

User control

In most interactions, a user should be able to undo and redo their actions. With most selection controls you can un-choose a selection, but in this instance you cannot click or tap a selected radio button to deselect it—selecting is a final action. The finality isn't conveyed when none are selected by default. Selecting a radio button by default communicates that the user is required to choose one in the set.

Expediting tasks

When one a choice in a set of radio buttons is the most desirable or frequently selected, it's helpful to select it by default. Doing this reduces the interaction cost and can save the user time and clicks.

The power of suggestion

Designs with a radio button selected by default make a strong suggestion to the user. It can help them make the best decision and increase their confidence. (Use this guidance with caution, and only for good.)

Development guidelines

Usage

Render a user interface to select the user type using radio inputs.

```
import { RadioControl } from '@wordpress/components';
import { useState } from 'react';

const MyRadioControl = () => {
    const [ option, setOption ] = useState( 'a' );

    return (
        <RadioControl
            label="User type"
            help="The type of the current user"
            selected={ option }
            options={ [
                { label: 'Author', value: 'a' },
                { label: 'Editor', value: 'e' },
            ] }
            onChange={ ( value ) => setOption( value ) }
        />
    );
};
```

Props

The component accepts the following props:

`help: string | Element`

If this property is added, a help text will be generated using help property as the content.

- Required: No

`hideLabelFromVision: boolean`

If true, the label will only be visible to screen readers.

- Required: No

`label: string`

If this property is added, a label will be generated using label property as the content.

- Required: No

`onChange: (value: string) => void`

A function that receives the value of the new option that is being selected as input.

- Required: Yes

`options: { label: string, value: string }[]`

An array of objects containing the value and label of the options.

- `label: string` The label to be shown to the user.
- `value: string` The internal value compared against select and passed to onChange.

- Required: No

`selected: string`

The value property of the currently selected option.

- Required: No

Related components

- To select one or more items from a set, use the `CheckboxControl` component.
- To toggle a single setting on or off, use the `ToggleControl` component.
- To format as a button group, use the `RadioGroup` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: RadioControl](#)

[Previous QueryControls](#) [Previous: QueryControls](#)
[Next RadioGroup](#) [Next: RadioGroup](#)

RadioGroup

In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
 - [Best practices](#)
 - [States](#)
- [Development guidelines](#)
 - [Usage](#)
- [Related components](#)

[↑ Back to top](#)

This component is deprecated. Consider using `RadioControl` or `ToggleGroupControl` instead. This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Use a RadioGroup component when you want users to select one option from a small set of options.



[Design guidelines](#)

[Usage](#)

Selected action

Only one option in a radio group can be selected and active at a time. Selecting one option deselects any other.

Best practices

Radio groups should:

- **Be clearly and accurately labeled.**
- **Clearly communicate that clicking or tapping will trigger an action.**
- **Use established colors appropriately.** For example, only use red buttons for actions that are difficult or impossible to undo.
- **Have consistent locations in the interface.**
- **Have a default option already selected.**

States

Active and available radio groups

A radio group's state makes it clear which option is active. Hover and focus states express the available selection options for buttons in a button group.

Disabled radio groups

Radio groups that cannot be selected can either be given a disabled state, or be hidden.

Development guidelines

Usage

Controlled

```
import { useState } from 'react';
import {
    __experimentalRadio as Radio,
    __experimentalRadioGroup as RadioGroup,
} from '@wordpress/components';

const MyControlledRadioRadioGroup = () => {
    const [ checked, setChecked ] = useState( '25' );
    return (
        <RadioGroup label="Width" onChange={ setChecked } checked={ checked }>
            <Radio value="25">25%</Radio>
            <Radio value="50">50%</Radio>
            <Radio value="75">75%</Radio>
            <Radio value="100">100%</Radio>
        </RadioGroup>
    );
}
```

Uncontrolled

When using the RadioGroup component as an uncontrolled component, the default value can be set with the `defaultChecked` prop.

```
import { useState } from 'react';
import {
  __experimentalRadio as Radio,
  __experimentalRadioGroup as RadioGroup,
} from '@wordpress/components';

const MyUncontrolledRadioRadioGroup = () => {
  return (
    <RadioGroup label="Width" defaultChecked="25">
      <Radio value="25">25%</Radio>
      <Radio value="50">50%</Radio>
      <Radio value="75">75%</Radio>
      <Radio value="100">100%</Radio>
    </RadioGroup>
  );
};


```

Related components

- For simple buttons that are related, use a `ButtonGroup` component.
- For traditional radio options, use a `RadioControl` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: RadioGroup](#)

[Previous RadioControl](#) [Previous: RadioControl](#)

[Next RangeControl](#) [Next: RangeControl](#)

RangeControl

In this article

Table of Contents

- [Design guidelines](#)
 - [Anatomy](#)
 - [Types](#)
 - [Behavior](#)
 - [Usage](#)
- [Development guidelines](#)
 - [Usage](#)
- [Props](#)
 - [afterIcon: string|Function|Component|null](#)

- [allowReset: boolean](#)
- [beforeIcon: string|Function|Component|null](#)
- [color: CSSProperties\['color'\]](#)
- [currentInput: number](#)
- [disabled: boolean](#)
- [help: string|Element](#)
- [hideLabelFromVision: boolean](#)
- [icon: string](#)
- [initialPosition: number](#)
- [isShiftStepEnabled: boolean](#)
- [label: string](#)
- [marks: Array](#)
- [onBlur: FocusEventHandler< HTMLInputElement >](#)
- [onChange: \(value?: number \) => void](#)
- [onFocus: FocusEventHandler< HTMLInputElement >](#)
- [onMouseLeave: MouseEventHandler< HTMLInputElement >](#)
- [onMouseMove: MouseEventHandler< HTMLInputElement >](#)
- [min: number](#)
- [max: number](#)
- [railColor: CSSProperties\['color' \]](#)
- [renderTooltipContent: \(value \) => value](#)
- [resetFallbackValue: number](#)
- [separatorType: 'none' | 'fullWidth' | 'topFullWidth'](#)
- [shiftStep: number](#)
- [showTooltip: boolean](#)
- [step: number | 'any'](#)
- [trackColor: CSSProperties\['color' \]](#)
- [type: string](#)
- [value: number](#)
- [withInputField: boolean](#)

- [Related components](#)

[↑ Back to top](#)

RangeControls are used to make selections from a range of incremental values.



Design guidelines

Anatomy

A RangeControl can contain the following elements:

1. **Rail:** The rail represents the entire surface area of the slider, from the minimum value selectable by the user to the maximum value selectable by the user. For left-to-right (LTR) languages, the minimum value appears on the far left, and the maximum value on the far right. For right-to-left (RTL) languages this orientation is reversed, with the minimum value on the far right and the maximum value on the far left.
2. **Track:** The track represents the portion of the rail from the minimum value to the currently selected value.
3. **Thumb:** The thumb slides along the track, displaying the selected value through its position.
4. **Value entry field:** The value entry field displays the currently selected, specific numerical value.
5. **Icon** (optional): An icon can be displayed before or after the slider.
6. **Tick mark** (optional): Tick marks represent predetermined values to which the user can move the slider.

Types

Continuous sliders

Continuous sliders allow users to select a value along a subjective range. They do not display the selected numeric value. Use them when displaying/editing the numeric value is not important, like volume.

Discrete sliders

Discrete sliders can be adjusted to a specific value by referencing its value entry field. Use them when it's important to display/edit the numeric value, like text size.

Possible selections may be organized through the use of tick marks, which a thumb will snap to (or to which an input will round up or down).

Behavior

- **Click and drag:** The slider is controlled by clicking the thumb and dragging it.
- **Click jump:** The slider is controlled by clicking the track.
- **Click and arrow:** The slider is controlled by clicking the thumb, then using arrow keys to move it.
- **Tab and arrow:** The slider is controlled by using the tab key to select the thumb of the desired slider, then using arrow keys to move it.
- **Value entry field:** Discrete sliders have value entry fields. After a text entry is made, the slider position automatically updates to reflect the new value.
- **Tick marks** (Optional) Discrete sliders can use evenly spaced tick marks along the slider track, and the thumb will snap to them. Each tick mark should change the setting in increments that are discernible to the user.

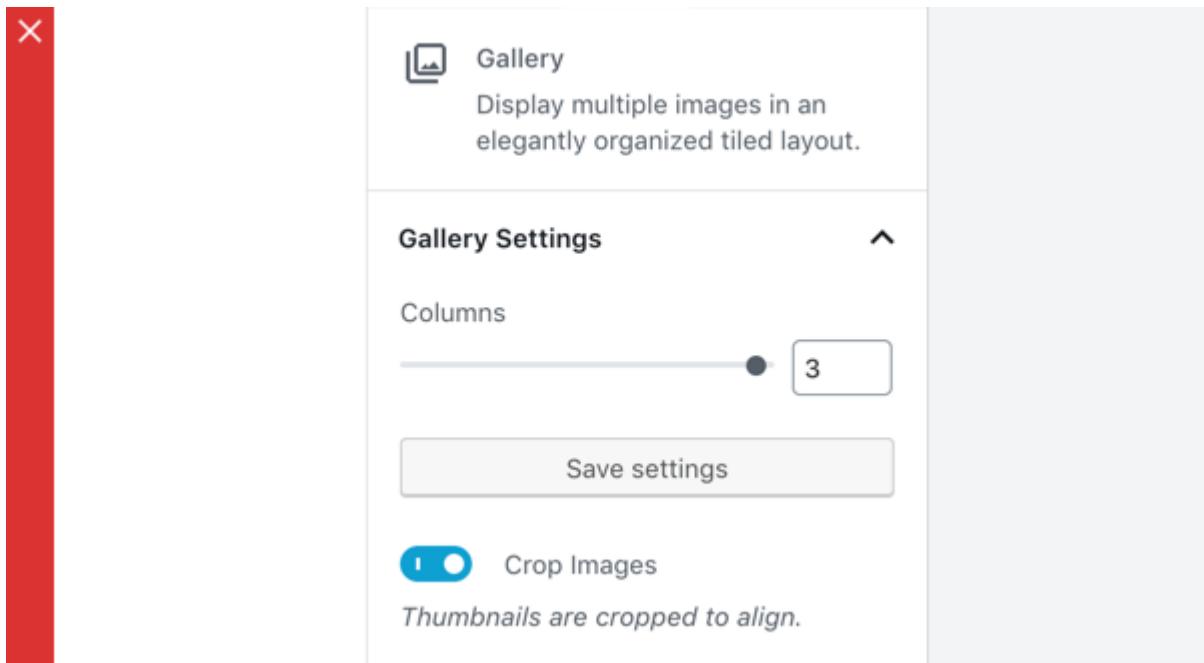
Usage

RangeControls reflect a range of values along a track, from which users may select a single value. They are ideal for adjusting settings such as volume, opacity, or text size.

RangeControls can have icons on both ends of the track that reflect a range of values.

Immediate effects

Changes made with RangeControls are immediate, allowing a user to make adjustments until finding their preference. They shouldn't be paired with settings that have delays in providing feedback.



Don't

Don't use RangeControls if the effect isn't immediate.

Current state

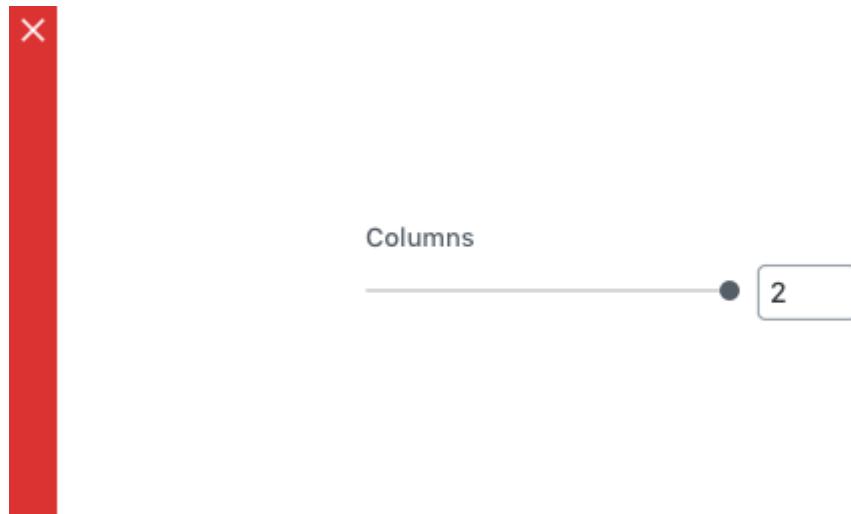
RangeControls reflect the current state of the settings they control.

Values



A RangeControl with an editable numeric value

Editable numeric values: Editable numeric values allow users to set the exact value of a RangeControl. After setting a value, the thumb position is immediately updated to match the new value.



Don't

RangeControls should only be used for choosing selections from a range of values (e.g., don't use a RangeControl if there are only 2 values).



Don't

RangeControls should provide the full range of choices available for the user to select from (e.g., don't disable only part of a RangeControl).

Development guidelines

Usage

Render a RangeControl to make a selection from a range of incremental values.

```
import { useState } from 'react';
import { RangeControl } from '@wordpress/components';

const MyRangeControl = () => {
    const [ columns, setColumns ] = useState( 2 );

    return(
        <RangeControl
            label="Columns"
            value={ columns }
            onChange={ ( value ) => setColumns( value ) }
            min={ 2 }
            max={ 10 }
        />
    );
};
```

Props

The set of props accepted by the component will be specified below. Props not included in this set will be applied to the input elements.

afterIcon: string|Function|Component|null

If this property is added, an [Icon component](#) will be rendered after the slider with the icon equal to `afterIcon`.

For more information on `IconType` see the [Icon component](#).

- Required: No
- Platform: Web

[allowReset: boolean](#)

If this property is true, a button to reset the slider is rendered.

- Required: No
- Default: `false`
- Platform: Web | Mobile

[beforeIcon: string|Function|Component|null](#)

If this property is added, an [Icon component](#) will be rendered before the slider with the icon equal to `beforeIcon`.

For more information on `IconType` see the [Icon component](#).

- Required: No
- Platform: Web

[color: CSSProperties\['color'\]](#)

CSS color string for the `RangeControl` wrapper.

- Required: No
- Platform: Web

[currentInput: number](#)

The current input to use as a fallback if `value` is currently `undefined`.

- Required: No
- Platform: Web

[disabled: boolean](#)

Disables the `input`, preventing new values from being applied.

- Required: No
- Platform: Web

[help: string|Element](#)

If this property is added, a help text will be generated using `help` property as the content.

- Required: No
- Platform: Web

[hideLabelFromVision: boolean](#)

Provides control over whether the label will only be visible to screen readers.

- Required: No

[icon: string](#)

An icon to be shown above the slider next to its container title.

- Required: No
- Platform: Mobile

[initialPosition: number](#)

The slider starting position, used when no value is passed. The initialPosition will be clamped between the provided min and max prop values.

- Required: No
- Platform: Web | Mobile

[isShiftStepEnabled: boolean](#)

Passed as a prop to the NumberControl component and is only applicable if withInputField is true. If true, while the number input has focus, pressing UP or DOWN along with the SHIFT key will change the value by the shiftStep value.

- Required: No

[label: string](#)

If this property is added, a label will be generated using label property as the content.

- Required: No
- Platform: Web | Mobile

[marks: Array|boolean](#)

Renders a visual representation of step ticks. Custom mark indicators can be provided by an Array.

Example:

```
const marks = [
  {
    value: 0,
    label: '0',
  },
  {
    value: 1,
    label: '1',
  },
  {
```

```
        value: 8,
        label: '8',
    },
{
    value: 10,
    label: '10',
},
];
};

const MyRangeControl() {
    return ( <RangeControl marks={ marks } min={ 0 } max={ 10 } step={ 1 } )
}
```

- Required: No
- Platform: Web

[onBlur: FocusEventHandler< HTMLInputElement >](#)

Callback for when RangeControl input loses focus.

- Required: No
- Platform: Web

[onChange: \(value?: number \)=> void](#)

A function that receives the new value. The value will be less than `max` and more than `min` unless a reset (enabled by `allowReset`) has occurred. In which case the value will be either that of `resetFallbackValue` if it has been specified or otherwise `undefined`.

- Required: No
- Platform: Web | Mobile

[onFocus: FocusEventHandler< HTMLInputElement >](#)

Callback for when RangeControl input gains focus.

- Required: No
- Platform: Web

[onMouseLeave: MouseEventHandler< HTMLInputElement >](#)

Callback for when mouse exits the RangeControl.

- Required: No
- Platform: Web

[onMouseMove: MouseEventHandler< HTMLInputElement >](#)

Callback for when mouse moves within the RangeControl.

- Required: No
- Platform: Web

[min: number](#)

The minimum value allowed.

- Required: No
- Default: 0
- Platform: Web | Mobile

[max: number](#)

The maximum value allowed.

- Required: No
- Default: 100
- Platform: Web | Mobile

[railColor: CSSProperties\['color' \]](#)

CSS color string to customize the rail element's background.

- Required: No
- Platform: Web

[renderTooltipContent: \(value \) => value](#)

A way to customize the rendered UI of the value. Example:

```
const customTooltipContent = value => `${value}%`  
  
const MyRangeControl() {  
    return (<RangeControl renderTooltipContent={ customTooltipContent } />  
}
```

- Required: No
- Platform: Web

[resetFallbackValue: number](#)

The value to revert to if the Reset button is clicked (enabled by allowReset)

- Required: No
- Platform: Web

[separatorType: 'none' | 'fullWidth' | 'topFullWidth'](#)

Define if separator line under/above control row should be disabled or full width. By default it is placed below excluding underline the control icon.

- Required: No
- Platform: Mobile

shiftStep: number

Passed as a prop to the NumberControl component and is only applicable if `withInputField` and `isShiftStepEnabled` are both true and while the number input has focus. Acts as a multiplier of `step`.

- Required: No

showTooltip: boolean

Forcing the Tooltip UI to show or hide. This is overridden to `false` when `step` is set to the special string value `any`.

- Required: No
- Platform: Web

step: number | ‘any’

The minimum amount by which `value` changes. It is also a factor in validation as `value` must be a multiple of `step` (offset by `min`) to be valid. Accepts the special string value `any` that voids the validation constraint and overrides both `withInputField` and `showTooltip` props to `false`.

- Required: No
- Platform: Web

trackColor: CSSProperties[‘color’]

CSS color string to customize the track element's background.

- Required: No
- Platform: Web

type: string

Define if the value selection should present a stepper control or a slider control in the bottom sheet on mobile. To use the stepper set the type value as `stepper`. Defaults to `slider` if no option is provided.

- Required: No
- Platform: Mobile

value: number

The current value of the range slider.

- Required: No
- Platform: Web | Mobile

[withInputField: boolean](#)

Determines if the `input` number field will render next to the RangeControl. This is overridden to `false` when `step` is set to the special string value `any`.

- Required: No
- Platform: Web

[Related components](#)

- To collect a numerical input in a text field, use the [TextControl](#) component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: RangeControl”](#)

[Previous RadioGroup](#) [Previous: RadioGroup](#)

[Next ResizableBox](#) [Next: ResizableBox](#)

ResizableBox

In this article

[Table of Contents](#)

- [Usage](#)
 - [Props](#)

[↑ Back to top](#)

ResizableBox is a wrapper around the [re-resizable package](#) with pre-defined classes and styles.

[Usage](#)

Most options are passed directly through to [re-resizable](#) so you may wish to refer to their documentation. The primary differences in this component are that we set `handleClasses` (to use custom class names) and pass some null values to `handleStyles` (to unset some inline styles).

The example below shows how you might use ResizableBox to set a width and height inside a block's edit component.

```

import { ResizableBox } from '@wordpress/components';

const Edit = ( props ) => {
    const {
        attributes: { height, width },
        setAttributes,
        toggleSelection,
    } = props;

    return (
        <ResizableBox
            size={ {
                height,
                width,
            } }
            minHeight="50"
            minWidth="50"
            enable={ {
                top: false,
                right: true,
                bottom: true,
                left: false,
                topRight: false,
                bottomRight: true,
                bottomLeft: false,
                topLeft: false,
            } }
            onResizeStop={ ( event, direction, elt, delta ) => {
                setAttributes( {
                    height: height + delta.height,
                    width: width + delta.width,
                } );
                toggleSelection( true );
            } }
            onResizeStart={ () => {
                toggleSelection( false );
            } }
        />
    );
};

```

Props

Name	Type	Default	Description
------	------	---------	-------------

showHandle bool false Determines if the resize handles are visible.

For additional props, check out [re-resizable](#).

First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: ResizableBox”](#)

[Previous RangeControl](#) [Previous: RangeControl](#)

[Next ResizeTooltip](#) [Next: ResizeTooltip](#)

ResizeTooltip

In this article

[Table of Contents](#)

- [Usage](#)
 - [Positions](#)
- [Props](#)
 - [axis](#)
 - [fadeTimeout](#)
 - [isVisible](#)
 - [labelRef](#)
 - [onMove](#)
 - [onResize](#)
 - [position](#)
 - [showPx](#)
 - [zIndex](#)

[↑ Back to top](#)

ResizeTooltip displays the dimensions of an element whenever the width or height of the element changes.

Usage

```
const Example = () => {
  return (
    <div style={{ position: 'relative' }}>
      <ResizeTooltip />
      ...
    </div>
  );
};
```

Be sure that the parent element containing `<ResizeTooltip />` has the `position` style property defined. This is important as `<ResizeTooltip />` uses position based techniques to determine size changes.

Positions

<ResizeTooltip /> has three positions;

- `bottom` (Default)
- `corner`

`bottom`

The `bottom` position (default) renders the dimensions label at the bottom-center of the (parent) element.

`corner`

The `corner` position renders the dimensions label in the top-right corner of the (parent) element.

Props

axis

Limits the label to render corresponding to the axis. By default, the label will automatically render based on both `x` and `y` changes.

- Type: `String`
- Required: No
- Values: `x` | `y`

fadeTimeout

Duration (in `ms`) before the label disappears after resize event.

- Type: `Number`
- Required: No
- Default: `180`

isVisible

Determines if the label can render.

- Type: `Boolean`
- Required: No
- Default: `true`

labelRef

Callback [Ref](#) for the label element.

- Type: `Function`
- Required: No

[onMove](#)

Callback function when the (observed) element resizes, specifically with a `mousemove` based event.

- Type: `Function`
- Required: No

[onResize](#)

Callback function when the (observed) element resizes.

- Type: `Function`
- Required: No

[position](#)

The positions for the label.

- Type: `String`
- Required: No
- Default: `corner`
- Values: `bottom|corner`

[showPx](#)

Renders a `PX` unit suffix after the width or height value in the label.

- Type: `Boolean`
- Required: No
- Default: `true`

[zIndex](#)

The `z-index` style property for the label.

- Type: `Number`
- Required: No
- Default: `1000`

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ResizeTooltip”](#)

ResponsiveWrapper

In this article

Table of Contents

- [Usage](#)
 - [Usage with SVG elements](#)
- [Props](#)
 - [children: React.ReactElement](#)
 - [isInline: boolean](#)
 - [naturalHeight: number](#)
 - [naturalWidth: number](#)

[↑ Back to top](#)

A wrapper component that maintains its aspect ratio when resized.

[Usage](#)

```
import { ResponsiveWrapper } from '@wordpress/components';

const MyResponsiveWrapper = () => (
  <ResponsiveWrapper naturalWidth={ 2000 } naturalHeight={ 680 }>
    
  </ResponsiveWrapper>
);
```

[Usage with SVG elements](#)

When passing an SVG element as the `<ResponsiveWrapper />`'s child, make sure that it has the `viewBox` and the `preserveAspectRatio` set.

When dealing with SVGs, it may not be possible to derive its `naturalWidth` and `naturalHeight` and therefore passing them as properties to `<ResponsiveWrapper />`. In this case, the SVG simply keeps scaling up to fill its container, unless the `height` and `width` attributes are specified.

[Props](#)

[children: React.ReactElement](#)

The element to wrap.

- Required: Yes

[isInline: boolean](#)

If true, the wrapper will be `span` instead of `div`.

- Required: No
- Default: `false`

[naturalHeight: number](#)

The intrinsic height of the element to wrap. Will be used to determine the aspect ratio.

- Required: No

[naturalWidth: number](#)

The intrinsic width of the element to wrap. Will be used to determine the aspect ratio.

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ResponsiveWrapper](#)

[Previous ResizeTooltip](#) [Previous: ResizeTooltip](#)

[Next Sandbox](#) [Next: Sandbox](#)

Sandbox

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [html: string](#)
 - [onFocus: React.DOMAttributes<HTMLIFrameElement>\[‘onFocus’\]](#)
 - [scripts: string\[\]](#)
 - [styles: string\[\]](#)
 - [title: string](#)
 - [type: string](#)
 - [tabIndex: HTMLElement\[‘tabIndex’\]](#)

[↑ Back to top](#)

This component provides an isolated environment for arbitrary HTML via iframes.

Usage

```
import { SandBox } from '@wordpress/components';

const MySandBox = () => (
  <SandBox html=<p>Content</p> title="SandBox" type="embed" />
);
```

Props

html: string

The HTML to render in the body of the iframe document.

- Required: No
- Default: “”

onFocus: React.DOMAttributes<HTMLIFrameElement>|‘onFocus’]

The onFocus callback for the iframe.

- Required: No

scripts: string[]

An array of script URLs to inject as <script> tags into the bottom of the <body> of the iframe document.

- Required: No
- Default: []

styles: string[]

An array of CSS strings to inject into the <head> of the iframe document.

- Required: No
- Default: []

title: string

The <title> of the iframe document.

- Required: No
- Default: “”

type: string

The CSS class name to apply to the <html> and <body> elements of the iframe.

- Required: No

- Default: “”

[tabIndex: HTMLElement\[‘tabIndex’ \]](#)

The `tabindex` the iframe should receive.

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Sandbox](#)”

[Previous ResponsiveWrapper](#) [Previous: ResponsiveWrapper](#)

[Next ScrollLock](#) [Next: ScrollLock](#)

ScrollLock

[↑ Back to top](#)

ScrollLock is a content-free React component for declaratively preventing scroll bleed from modal UI to the page body. This component applies a `lockscroll` class to the `document.documentElement` and `document.scrollingElement` elements to stop the body from scrolling. When it is present, the lock is applied.

Usage

Declare scroll locking as part of modal UI.

```
import { useState } from 'react';
import { ScrollLock, Button } from '@wordpress/components';

const MyScrollLock = () => {
    const [ isScrollLocked, setIsScrollLocked ] = useState( false );

    const toggleLock = () => {
        setIsScrollLocked( ( locked ) => ! locked );
    };

    return (
        <div>
            <Button variant="secondary" onClick={ toggleLock }>
                Toggle scroll lock
            </Button>
        </div>
    );
}
```

```
        </Button>
        { isScrollLocked && <ScrollLock /> }
        <p>
          Scroll locked:
          <strong>{ isScrollLocked ? 'Yes' : 'No' }</strong>
        </p>
      </div>
    );
}
};
```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ScrollLock”](#)

[Previous Sandbox](#) [Previous: Sandbox](#)
[Next Scrollable](#) [Next: Scrollable](#)

Scollable

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [children: ReactNode](#)
 - [scrollDirection: string](#)
 - [smoothScroll: boolean](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`Scollable` is a layout component that content in a scrollable container.

[Usage](#)

```
import { __experimentalScollable as Scollable } from '@wordpress/compone

function Example() {
  return (
    <Scollable style={{ maxHeight: 200 }}>
```

```
        <div style={ { height: 500 } }>...</div>
      </Scrollable>
    );
}
```

Props

children: ReactNode

The children elements.

- Required: Yes

scrollDirection: string

Renders a scrollbar for a specific axis when content overflows.

- Required: No
- Default: y
- Allowed values: x, y, auto

smoothScroll: boolean

Enables (CSS) smooth scrolling.

- Required: No
- Default: false

First published

June 7, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Scrollable](#)

[Previous ScrollLock](#) [Previous: ScrollLock](#)
[Next SearchControl](#) [Next: SearchControl](#)

SearchControl

In this article

Table of Contents

- [Development guidelines](#)
 - [Usage](#)

- [Props](#)
- [hideLabelFromVision](#)
- [Related components](#)

[↑ Back to top](#)

SearchControl components let users display a search control.

Check out the [Storybook page](#) for a visual exploration of this component.

Development guidelines

Usage

Render a user interface to input the name of an additional css class.

```
import { useState } from 'react';
import { __ } from '@wordpress/i18n';
import { SearchControl } from '@wordpress/components';

function MySearchControl( { className, setState } ) {
    const [ searchInput, setSearchInput ] = useState( '' );

    return (
        <SearchControl
            label={__( 'Search posts' ) }
            value={searchInput}
            onChange={setSearchInput}
        />
    );
}
```

Props

The set of props accepted by the component will be specified below.
Props not included in this set will be applied to the input element.

label

If this property is added, a label will be generated using label property as the content.

A label should always be provided as an accessibility best practice, even when a placeholder is defined
and `hideLabelFromVision` is `true`.

- Type: `String`
- Required: Yes

placeholder

If this property is added, a specific placeholder will be used for the input.

- Type: `String`
- Required: No
- Default: `__('Search')`

value

The current value of the input.

- Type: `String`
- Required: No

className

The class that will be added to the classes of the wrapper div.

- Type: `String`
- Required: No

onChange

A function that receives the value of the input.

- Type: `function`
- Required: Yes

help

If this property is added, a help text will be generated using help property as the content.

- Type: `String|Element`
- Required: No

hideLabelFromVision

If true, the label will not be visible, but will be read by screen readers. Defaults to `true`.

- Type: `Boolean`
- Required: No

`size: 'default' | 'compact'`

The size of the component.

- Required: No
- Default: `'default'`

Related components

- To offer users more constrained options for input, use TextControl, SelectControl, RadioControl, CheckboxControl, or RangeControl.

First published

July 7, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: SearchControl”](#)

[Previous Scrollable](#) [Previous: Scrollable](#)
[Next SelectControl](#) [Next: SelectControl](#)

SelectControl

In this article

Table of Contents

- [Design guidelines](#)
 - [Usage](#)
 - [Behavior](#)
 - [Content Guidelines](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
 - [nextHasNoMarginBottom](#)
- [Related components](#)

[↑ Back to top](#)

SelectControl allow users to select from a single or multiple option menu. It functions as a wrapper around the browser’s native `<select>` element.

Link To

None

Design guidelines

Usage

When to use a select control

Use a select control when:

- You want users to select one or more options from a list.
- There is a strong default option.
- There is little available space.
- The contents of the hidden part of the menu are obvious from its label and the one selected item. For example, if you have an option menu labelled “Month:” with the item “January” selected, the user might reasonably infer that the menu contains the 12 months of the year without having to look.

If you have a shorter list of options, consider using RadioControl instead.



Post Format

A
G
L
I
Q
✓ S
V
A

Do

Use selects when you have multiple options.



Crop images?

- Yes
- No

Don't

Use selects for binary questions.

Behavior

A SelectControl includes a double-arrow indicator. The menu appears layered over the select.

Opening and Closing

Once the menu is displayed onscreen, it remains open until the user chooses a menu item, clicks outside of the menu, or switches to another browser tab.

Content Guidelines

Labels

Label the SelectControl with a text label above it, or to its left, using sentence capitalization. Clicking the label allows the user to focus directly on the select.



Link to

None

Post Format

Do

Position the label above, or to the left of, the select.



Link to

None

Post Format

Don't

Position the label centered over the select, or right aligned against the side of the select.

Menu Items

- Menu items should be short — ideally, single words — and use sentence capitalization.
- Do not use full sentences inside menu items.
- Ensure that menu items are ordered in a way that is most useful to users. Alphabetical or recency ordering is preferred.



Link to

Attachment Page

Media File

✓ None

Do

Use short menu items.



Link to

- ✓ Link to the image's attachment
- Link to the image's media item
- Don't link to anything.

Don't

Use sentences in your menu.

Development guidelines

Usage

Render a user interface to select the size of an image.

```
import { useState } from 'react';
import { SelectControl } from '@wordpress/components';

const MySelectControl = () => {
    const [ size, setSize ] = useState( '50%' );

    return (
        <SelectControl
            label="Size"
            value={ size }
            options={ [
                { label: 'Big', value: '100%' },
                { label: 'Medium', value: '50%' },
                { label: 'Small', value: '25%' },
            ] }
        />
    );
}
```

```

        ] }
      onChange={ ( newSize ) => setSize( newSize ) }
      __nextHasNoMarginBottom
    />
  );
};


```

Render a user interface to select multiple users from a list.

```

<SelectControl
  multiple
  label={ __( 'Select some users:' ) }
  value={ this.state.users } // e.g: value = [ 'a', 'c' ]
  onChange={ ( users ) => {
    this.setState( { users } );
  } }
  options={ [
    { value: '', label: 'Select a User', disabled: true },
    { value: 'a', label: 'User A' },
    { value: 'b', label: 'User B' },
    { value: 'c', label: 'User c' },
  ] }
  __nextHasNoMarginBottom
/>

```

Render a user interface to select items within groups

```

const [ item, setItem ] = useState( '' );

// ...

<SelectControl
  label={ __( 'Select an item:' ) }
  value={ item } // e.g: value = 'a'
  onChange={ ( selection ) => { setItem( selection ) } }
  __nextHasNoMarginBottom
>
  <optgroup label="Theropods">
    <option value="Tyrannosaurus">Tyrannosaurus</option>
    <option value="Velociraptor">Velociraptor</option>
    <option value="Deinonychus">Deinonychus</option>
  </optgroup>
  <optgroup label="Sauropods">
    <option value="Diplodocus">Diplodocus</option>
    <option value="Saltasaurus">Saltasaurus</option>
    <option value="Apatosaurus">Apatosaurus</option>
  </optgroup>
</SelectControl>

```

Props

- The set of props accepted by the component will be specified below.
- Props not included in this set will be applied to the select element.

- One important prop to refer is `value`. If `multiple` is `true`, `value` should be an array with the values of the selected options.
- If `multiple` is `false`, `value` should be equal to the value of the selected option.

label

If this property is added, a label will be generated using label property as the content.

- Type: `String`
- Required: No

labelPosition

The position of the label (`top`, `side`, or `bottom`).

- Type: `String`
- Required: No

hideLabelFromVision

If true, the label will only be visible to screen readers.

- Type: `Boolean`
- Required: No

help

If this property is added, a help text will be generated using help property as the content.

- Type: `String | Element`
- Required: No

multiple

If this property is added, multiple values can be selected. The `value` passed should be an array.

In most cases, it is preferable to use the `FormTokenField` or `CheckboxControl` components instead.

- Type: `Boolean`
- Required: No

options

An array of objects containing the following properties:

- `label`: (string) The label to be shown to the user.
- `value`: (string) The internal value used to choose the selected value. This is also the value passed to `onChange` when the option is selected.
- `disabled`: (boolean) Whether or not the option should have the disabled attribute.
- Type: `Array`
- Required: No

children

An alternative to the `options` prop.

Use the `children` prop to have more control on the style of the items being rendered, like `optgroup`s or `option`s and possibly avoid re-rendering due to the reference update on the `options` prop.

- Type: `ReactNode`

- Required: No

onChange

A function that receives the value of the new option that is being selected as input.

If `multiple` is true the value received is an array of the selected value.

If `multiple` is false the value received is a single value with the new selected value.

- Type: `function`
- Required: Yes

[nextHasNoMarginBottom](#)

Start opting into the new margin-free styles that will become the default in a future version.

- Type: `Boolean`
- Required: No
- Default: `false`

[Related components](#)

- To select one option from a set, and you want to show them all the available options at once, use the `Radio` component.
- To select one or more items from a set, use the `CheckboxControl` component.
- To toggle a single setting on or off, use the `ToggleControl` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: SelectControl](#)

[Previous SearchControl](#) [Previous: SearchControl](#)

[Next SlotFill](#) [Next: SlotFill](#)

SlotFill

In this article

Table of Contents

- [Usage](#)
- [Props](#)

[↑ Back to top](#)

Slot and Fill are a pair of components which enable developers to render elsewhere in a React element tree, a pattern often referred to as “portal” rendering. It is a pattern for component extensibility, where a single Slot may be occupied by an indeterminate number of Fills elsewhere in the application.

Slot Fill is heavily inspired by the [react-slot-fill library](#), but uses React’s own portal rendering API.

Usage

At the root of your application, you must render a `SlotFillProvider` which coordinates Slot and Fill rendering.

Then, render a Slot component anywhere in your application, giving it a name.

Any Fill will automatically occupy this Slot space, even if rendered elsewhere in the application.

You can either use the Fill component directly, or a wrapper component type as in the below example to abstract the slot name from consumer awareness.

```
import {  
    SlotFillProvider,  
    Slot,  
    Fill,  
    Panel,  
    PanelBody,  
} from '@wordpress/components';  
  
const MySlotFillProvider = () => {  
    const MyPanelSlot = () => (  
        <Panel header="Panel with slot">  
            <PanelBody>  
                <Slot name="MyPanelSlot" />  
            </PanelBody>  
        </Panel>  
    );  
  
    MyPanelSlot.Content = () => <Fill name="MyPanelSlot">Panel body</Fill>  
  
    return (  
        <SlotFillProvider>  
            <MyPanelSlot />  
        </SlotFillProvider>  
    );  
};
```

```

        <SlotFillProvider>
            <MyPanelSlot />
            <MyPanelSlot.Content />
        </SlotFillProvider>
    );
}

```

There is also `createSlotFill` helper method which was created to simplify the process of matching the corresponding `Slot` and `Fill` components:

```

const { Fill, Slot } = createSlotFill( 'Toolbar' );

const ToolbarItem = () => <Fill>My item</Fill>

const Toolbar = () => (
    <div className="toolbar">
        <Slot />
    </div>
);

```

Props

The `SlotFillProvider` component does not accept any props.

Both `Slot` and `Fill` accept a `name` string prop, where a `Slot` with a given `name` will render the `children` of any associated `Fills`.

`Slot` accepts a `bubblesVirtually` prop which changes the event bubbling behaviour:

- By default, events will bubble to their parents on the DOM hierarchy (native event bubbling)
- If `bubblesVirtually` is set to true, events will bubble to their virtual parent in the React elements hierarchy instead.

`Slot` with `bubblesVirtually` set to true also accept optional `className` and `style` props to add to the slot container.

`Slot` **without** `bubblesVirtually` accepts an optional `children` function prop, which takes `fills` as a param. It allows you to perform additional processing and wrap `fills` conditionally.

Example:

```

const Toolbar = ( { isMobile } ) => (
    <div className="toolbar">
        <Slot name="Toolbar">
            { ( fills ) => {
                return isMobile && fills.length > 3 ? (
                    <div className="toolbar__mobile-long">{ fills }</div>
                ) : (
                    fills
                );
            }
        </Slot>
)

```

```
</div>
);

Props can also be passed from a Slot to a Fill by using the prop fillProps on the Slot:

const { Fill, Slot } = createSlotFill( 'Toolbar' );

const ToolbarItem = () => (
  <Fill>
    { ( { hideToolbar } ) => {
      <Button onClick={ hideToolbar }>Hide</Button>;
    } }
  </Fill>
);

const Toolbar = () => {
  const hideToolbar = () => {
    console.log( 'Hide toolbar' );
  };
  return (
    <div className="toolbar">
      <Slot fillProps={ { hideToolbar } } />
    </div>
  );
};


```

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: SlotFill](#)

[Previous SelectControl](#) [Previous: SelectControl](#)

[Next Snackbar](#) [Next: Snackbar](#)

Snackbar

In this article

Table of Contents

- [Design guidelines](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

Use Snackbars to communicate low priority, non-interruptive messages to the user.

Design guidelines

A Snackbar displays a succinct message that is cleared out after a small delay. It can also offer the user options, like viewing a published post but these options should also be available elsewhere in the UI.

Development guidelines

Usage

To display a plain snackbar, pass the message as a `children` prop:

```
const MySnackbarNotice = () => (
  <Snackbar>Post published successfully.</Snackbar>
);
```

For more complex markup, you can pass any JSX element:

```
const MySnackbarNotice = () => (
  <Snackbar>
    <p>
      An error occurred: <code>{ errorDetails }</code>.
    </p>
  </Snackbar>
);
```

Props

The following props are used to control the display of the component.

`actions: NoticeAction[]`

An array of action objects. Each member object should contain a `label` and either a `url` link string or `onClick` callback function.

- Required: No
- Default: []

`children: string`

The displayed message of a notice. Also used as the spoken message for assistive technology, unless `spokenMessage` is provided as an alternative message.

- Required: Yes

`explicitDismiss: boolean`

Whether to require user action to dismiss the snackbar. By default, this is dismissed on a timeout, without user interaction.

- Required: No
- Default: `false`

`icon: ReactNode`

The icon to render in the snackbar.

- Required: No
- Default: `null`

`listRef: MutableRefObject< HTMLDivElement | null >`

A ref to the list that contains the snackbar.

- Required: No

`onDismiss: () => void`

A callback executed when the snackbar is dismissed. It is distinct from `onRemove`, which *looks* like a callback but is actually the function to call to remove the snackbar from the UI.

- Required: No

`onRemove: () => void`

Function called when dismissing the notice.

- Required: No

`politeness: 'polite' | 'assertive'`

A politeness level for the notice's spoken message. Should be provided as one of the valid options for [an aria-live attribute value](#). Note that this value should be considered a suggestion; assistive technologies may override it based on internal heuristics.

A value of '`assertive`' is to be used for important, and usually time-sensitive, information. It will interrupt anything else the screen reader is announcing in that moment.

A value of '`polite`' is to be used for advisory information. It should not interrupt what the screen reader is announcing in that moment (the “speech queue”) or interrupt the current task.

- Required: No
- Default: '`polite`'

`spokenMessage: string`

Used to provide a custom spoken message.

- Required: No
- Default: `children`

Related components

- To create a prominent message that requires a higher-level of attention, use a Notice.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Snackbar”](#)

[Previous SlotFill](#) [Previous: SlotFill](#)

[Next Spacer](#) [Next: Spacer](#)

Spacer

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [children: ReactNode](#)
 - [margin: number](#)
 - [marginBottom: number](#)
 - [marginLeft: number](#)
 - [marginRight: number](#)
 - [marginTop: number](#)
 - [marginX: number](#)
 - [marginY: number](#)
 - [padding: number](#)
 - [paddingBottom: number](#)
 - [paddingLeft: number](#)
 - [paddingRight: number](#)
 - [paddingTop: number](#)
 - [paddingX: number](#)
 - [paddingY: number](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`Spacer` is a primitive layout component that provides inner (`padding`) or outer (`margin`) space in-between components. It can also be used to adaptively provide space within an `HStack` or `VStack`.

Usage

`Spacer` comes with a bunch of shorthand props to adjust `margin` and `padding`. The values of these props work as a multiplier to the library’s grid system (base of 4px).

```
import {
  __experimentalSpacer as Spacer,
  __experimentalHeading as Heading,
  __experimentalView as View,
} from '@wordpress/components';

function Example() {
  return (
    <View>
      <Spacer>
        <Heading>WordPress.org</Heading>
      </Spacer>
      <Text>Code is Poetry</Text>
    </View>
  );
}
```

Props

children: ReactNode

The children elements.

- Required: No

margin: number

Adjusts all margins.

- Required: No

marginBottom: number

Adjusts bottom margin, potentially overriding the value from the more generic `margin` and `marginY` props.

- Required: No
- Default: 2

marginLeft: number

Adjusts left margin, potentially overriding the value from the more generic `margin` and `marginX` props.

- Required: No

marginRight: number

Adjusts right margin, potentially overriding the value from the more generic `margin` and `marginX` props.

- Required: No

marginTop: number

Adjusts top margin, potentially overriding the value from the more generic `margin` and `marginY` props.

- Required: No

marginX: number

Adjusts left and right margins, potentially overriding the value from the more generic `margin` prop.

- Required: No

marginY: number

Adjusts top and bottom margins, potentially overriding the value from the more generic `margin` prop.

- Required: No

padding: number

Adjusts all padding.

- Required: No

paddingBottom: number

Adjusts bottom padding, potentially overriding the value from the more generic `padding` and `paddingY` props.

- Required: No

[paddingLeft: number](#)

Adjusts left padding, potentially overriding the value from the more generic `padding` and `paddingX` props.

- Required: No

[paddingRight: number](#)

Adjusts right padding, potentially overriding the value from the more generic `padding` and `paddingX` props.

- Required: No

[paddingTop: number](#)

Adjusts top padding, potentially overriding the value from the more generic `padding` and `paddingY` props.

- Required: No

[paddingX: number](#)

Adjusts left and right padding, potentially overriding the value from the more generic `padding` prop.

- Required: No

[paddingY: number](#)

Adjusts top and bottom padding, potentially overriding the value from the more generic `padding` prop.

- Required: No

First published

May 6, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Spacer”](#)

[Previous Snackbar](#) [Previous: Snackbar](#)

[Next Spinner](#) [Next: Spinner](#)

Spinner

In this article

[Table of Contents](#)

- [Usage](#)
- [Best practices](#)

[↑ Back to top](#)

Spinner is a component used to notify users that their action is being processed.

[Usage](#)

```
import { Spinner } from '@wordpress/components';

function Example() {
    return <Spinner />;
}
```

[Best practices](#)

The spinner component should:

- Signal to users that the processing of their request is underway and will soon complete.

Check out the [Storybook page](#) for a visual exploration of this component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Spinner”](#)

[Previous Spacer](#) [Previous: Spacer](#)
[Next Surface](#) [Next: Surface](#)

Surface

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [backgroundSize: number](#)
 - [borderBottom: boolean](#)
 - [borderLeft: boolean](#)
 - [borderRight: boolean](#)
 - [borderTop: boolean](#)
 - [variant: string](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`Surface` is a core component that renders a primary background color.

[Usage](#)

In the example below, notice how the `Surface` renders in white (or dark gray if in dark mode).

```
import {  
    __experimentalSurface as Surface,  
    __experimentalText as Text,  
} from '@wordpress/components';  
  
function Example() {  
    return (  
        <Surface>  
            <Text>Code is Poetry</Text>  
        </Surface>  
    );  
}
```

[Props](#)

[backgroundSize: number](#)

- Required: No
- Default: 12

Determines the grid size for “dotted” and “grid” variants.

[borderBottom: boolean](#)

- Required: No
- Default: `false`

Renders a bottom border.

[borderLeft: boolean](#)

- Required: No
- Default: `false`

Renders a left border.

[borderRight: boolean](#)

- Required: No
- Default: `false`

Renders a right border.

[borderTop: boolean](#)

- Required: No
- Default: `false`

Renders a top border.

[variant: string](#)

- Required: No
- Default: `false`
- Allowed values: `primary`, `secondary`, `tertiary`, `dotted`, `grid`

Modifies the background color of Surface.

- `primary`: Used for almost all cases.
- `secondary`: Used as a secondary background for inner Surface components.
- `tertiary`: Used as the app/site wide background. Visible in **dark mode** only. Use case is rare.
- `grid`: Used to show a grid.
- `dotted`: Used to show a dots grid.

First published

June 8, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Surface”](#)

[Previous Spinner](#) [Previous: Spinner](#)

[Next TabPanel](#) [Next: TabPanel](#)

TabPanel

In this article

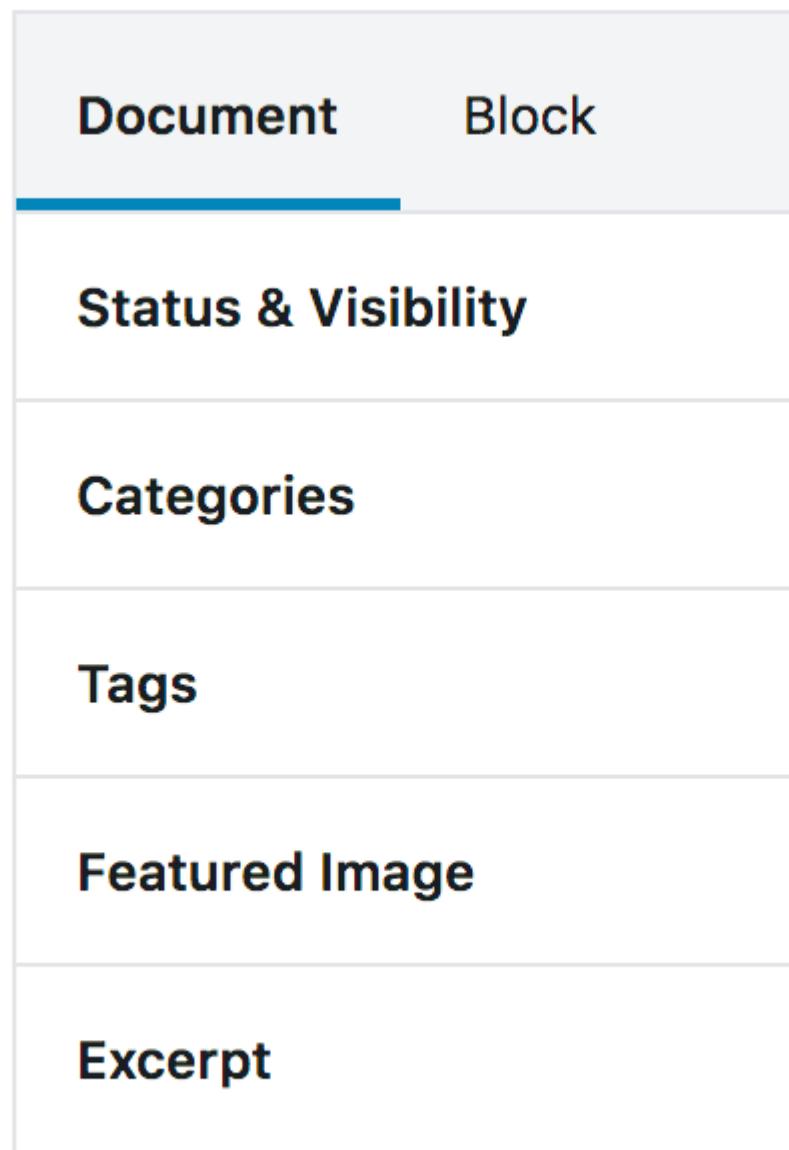
Table of Contents

- [Design guidelines](#)
 - [Usage](#)
 - [Anatomy](#)
 - [Behavior](#)
 - [Placement](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)

[↑ Back to top](#)

TabPanel is a React component to render an ARIA-compliant TabPanel.

TabPanels organize content across different screens, data sets, and interactions. It has two sections: a list of tabs, and the view to show when tabs are chosen.



Design guidelines

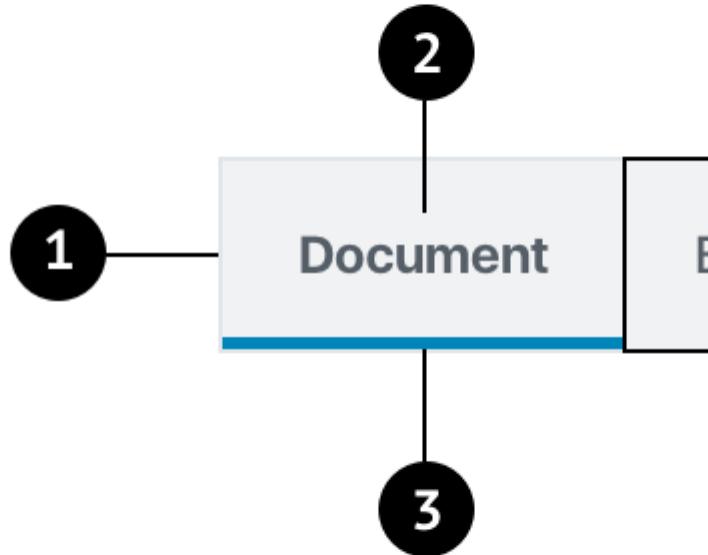
Usage

TabPanels organize and allow navigation between groups of content that are related and at the same level of hierarchy.

Tabs in a set

As a set, all tabs are unified by a shared topic. For clarity, each tab should contain content that is distinct from all the other tabs in a set.

Anatomy



1. Container
2. Active text label
3. Active tab indicator
4. Inactive text label
5. Tab item

Labels

Tab labels appear in a single row, in the same typeface and size. Use text labels that clearly and succinctly describe the content of a tab, and make sure that a set of tabs contains a cohesive group of items that share a common characteristic.

Tab labels can wrap to a second line, but do not add a second row of tabs.

Active tab indicators

To differentiate an active tab from an inactive tab, apply an underline and color change to the active tab's text and icon.

Document

Block

Behavior

Users can navigate between tabs by tapping the tab key on the keyboard.

Placement

Place tabs above content. Tabs control the UI region displayed below them.

Development guidelines

Usage

```
import { TabPanel } from '@wordpress/components';

const onSelect = ( tabName ) => {
    console.log( 'Selecting tab', tabName );
};

const MyTabPanel = () => (
    <TabPanel
        className="my-tab-panel"
        activeClass="active-tab"
        onSelect={ onSelect }
)
```

```

    tabs={ [
      {
        name: 'tab1',
        title: 'Tab 1',
        className: 'tab-one',
      },
      {
        name: 'tab2',
        title: 'Tab 2',
        className: 'tab-two',
      },
    ] }
  >
  { ( tab ) => <p>{ tab.title }</p> }
</TabPanel>
);

```

Props

The component accepts the following props:

className

The class to give to the outer container for the TabPanel

- Type: `String`
- Required: No
- Default: `”`

orientation

The orientation of the tablist (`vertical` or `horizontal`)

- Type: `String`
- Required: No
- Default: `horizontal`

onSelect

The function called when a tab has been selected. It is passed the `tabName` as an argument.

- Type: `Function`
- Required: No
- Default: `noop`

tabs

An array of objects containing the following properties:

- `name: (string)` Defines the key for the tab.
- `title:(string)` Defines the translated text for the tab.
- `className:(string)` Optional. Defines the class to put on the tab.

- `icon:(ReactNode)` Optional. When set, displays the icon in place of the tab title. The title is then rendered as an aria-label and tooltip.
- `disabled:(boolean)` Optional. Determines if the tab should be disabled or selectable.

Note: Other fields may be added to the object and accessed from the child function if desired.

- Type: `Array`
- Required: Yes

activeClass

The class to add to the active tab

- Type: `String`
- Required: No
- Default: `is-active`

initialTabName

The name of the tab to be selected upon mounting of component. If this prop is not set, the first tab will be selected by default.

- Type: `String`
- Required: No
- Default: `none`

selectOnMove

When `true`, the tab will be selected when receiving focus (automatic tab activation). When `false`, the tab will be selected only when clicked (manual tab activation). See the [official W3C docs](#) for more info.

- Type: `boolean`
- Required: No
- Default: `true`

children

A function which renders the tabviews given the selected tab. The function is passed the active tab object as an argument as defined the tabs prop.

- Type: `(Object) => Element`
- Required: Yes

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: TabPanel](#)

[Previous Surface](#) [Previous: Surface](#)

[Next Tabs](#) [Next: Tabs](#)

Tabs

In this article

Table of Contents

- [Development guidelines](#)
 - [Usage](#)
 - [Components and Sub-components](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Tabs is a collection of React components that combine to render an [ARIA-compliant tabs pattern](#).

Tabs organizes content across different screens, data sets, and interactions. It has two sections: a list of tabs, and the view to show when tabs are chosen.

[Development guidelines](#)

[Usage](#)

Uncontrolled Mode

Tabs can be used in an uncontrolled mode, where the component manages its own state. In this mode, the `initialTabId` prop can be used to set the initially selected tab. If this prop is not set, the first tab will be selected by default. In addition, in most cases where the currently active tab becomes disabled or otherwise unavailable, uncontrolled mode will automatically fall back to selecting the first available tab.

```
import { Tabs } from '@wordpress/components';

const onSelect = ( tabName ) => {
    console.log( 'Selecting tab', tabName );
};

const MyUncontrolledTabs = () => (
    <Tabs onSelect={onSelect} initialTab="tab2">
        <Tabs.TabList>
            <Tabs.Tab id={ 'tab1' } title={ 'Tab 1' }>
                Tab 1
            </Tabs.Tab>
            <Tabs.Tab id={ 'tab2' } title={ 'Tab 2' }>
```

```

        Tab 2
    </Tabs.Tab>
    <Tabs.Tab id={'tab3'} title={'Tab 3'}>
        Tab 3
    </Tabs.Tab>
</Tabs.TabList>
<Tabs.TabPanel id={'tab1'}>
    <p>Selected tab: Tab 1</p>
</Tabs.TabPanel>
<Tabs.TabPanel id={'tab2'}>
    <p>Selected tab: Tab 2</p>
</Tabs.TabPanel>
<Tabs.TabPanel id={'tab3'}>
    <p>Selected tab: Tab 3</p>
</Tabs.TabPanel>
</Tabs>
);

```

Controlled Mode

Tabs can also be used in a controlled mode, where the parent component specifies the `selectedTabId` and the `onSelect` props to control tab selection. In this mode, the `initialTabId` prop will be ignored if it is provided. If the `selectedTabId` is `null`, no tab is selected. In this mode, if the currently selected tab becomes disabled or otherwise unavailable, the component will *not* fall back to another available tab, leaving the controlling component in charge of implementing the desired logic.

```

import { Tabs } from '@wordpress/components';
const [ selectedTabId, setSelectedTabId ] = useState<
    string | undefined | null
>();

const onSelect = ( tabName ) => {
    console.log( 'Selecting tab', tabName );
};

const MyControlledTabs = () => (
    <Tabs
        selectedTabId={ selectedTabId }
        onSelect={ ( selectedId ) => {
            setSelectedTabId( selectedId );
            onSelect( selectedId );
        } }
    >
        <Tabs.TabList >
            <Tabs.Tab id={'tab1'} title={'Tab 1'}>
                Tab 1
            </Tabs.Tab>
            <Tabs.Tab id={'tab2'} title={'Tab 2'}>
                Tab 2
            </Tabs.Tab>
            <Tabs.Tab id={'tab3'} title={'Tab 3'}>
                Tab 3
            </Tabs.Tab>
        </Tabs.TabList >
    
```

```

        </Tabs.Tab>
    </Tabs.TabList>
    <Tabs.TabPanel id={ 'tab1' }>
        <p>Selected tab: Tab 1</p>
    </Tabs.TabPanel>
    <Tabs.TabPanel id={ 'tab2' }>
        <p>Selected tab: Tab 2</p>
    </Tabs.TabPanel>
    <Tabs.TabPanel id={ 'tab3' }>
        <p>Selected tab: Tab 3</p>
    </Tabs.TabPanel>
</Tabs>
);

```

Components and Sub-components

Tabs is comprised of four individual components:

- **Tabs**: a wrapper component and context provider. It is responsible for managing the state of the tabs and rendering the **TabList** and **TabPanels**.
- **TabList**: a wrapper component for the **Tab** components. It is responsible for rendering the list of tabs.
- **Tab**: renders a single tab. The currently active tab receives default styling that can be overridden with CSS targeting [aria-selected=”true”].
- **TabPanel**: renders the content to display for a single tab once that tab is selected.

Tabs

Props

`children: React.ReactNode`

The children elements, which should be at least a `Tabs.Tablist` component and a series of `Tabs.TabPanel` components.

- Required: Yes

`selectOnMove: boolean`

When `true`, the tab will be selected when receiving focus (automatic tab activation). When `false`, the tab will be selected only when clicked (manual tab activation). See the [official W3C docs](#) for more info.

- Required: No
- Default: `true`

`initialTabId: string`

The id of the tab to be selected upon mounting of component. If this prop is not set, the first tab will be selected by default. The id provided will be internally prefixed with a unique instance ID to avoid collisions.

Note: this prop will be overridden by the selectedTabId prop if it is provided. (Controlled Mode)

- Required: No

```
onSelect: ( ( selectedId: string | null | undefined ) => void )
```

The function called when a tab has been selected. It is passed the selected tab's ID as an argument.

- Required: No
- Default: noop

```
orientation: horizontal | vertical
```

The orientation of the tablist (vertical or horizontal)

- Required: No
- Default: horizontal

```
selectedTabId: string | null
```

The ID of the tab to display. This id is prepended with the Tabs instanceId internally. If left undefined, the component assumes it is being used in uncontrolled mode. Consequently, any value different than undefined will set the component in controlled mode. When in controlled mode, the null value will result in no tab being selected.

- Required: No

TabList

Props

```
children: React.ReactNode
```

The children elements, which should be a series of Tabs.TabPanel components.

- Required: No

Tab

Props

```
tabId: string
```

A unique identifier for the tab, which is used to generate a unique id for the underlying element. The value of this prop should match with the value of the tabId prop on the corresponding Tabs.TabPanel component.

- Required: Yes

```
children: React.ReactNode
```

The children elements, generally the text to display on the tab.

- Required: No

```
disabled: boolean
```

Determines if the tab button should be disabled.

- Required: No
- Default: `false`

```
render: React.ReactNode
```

The type of component to render the tab button as. If this prop is not provided, the tab button will be rendered as a `button` element.

- Required: No

TabPanel

Props

```
children: React.ReactNode
```

The children elements, generally the content to display on thetabpanel.

- Required: No

```
tabId: string
```

A unique identifier for thetabpanel, which is used to generate an instanced id for the underlying element. The value of this prop should match with the value of the `tabId` prop on the corresponding `Tabs.Tab` component.

- Required: Yes

```
focusable: boolean
```

Determines whether or not thetabpanel element should be focusable. If `false`, pressing the tab key will skip over thetabpanel, and instead focus on the first focusable element in the panel (if there is one).

- Required: No
- Default: `true`

First published

October 6, 2023

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: Tabs”](#)

[Previous TabPanel](#) [Previous: TabPanel](#)
[Next TextControl](#) [Next: TextControl](#)

TextControl

In this article

[Table of Contents](#)

- [Design guidelines](#)
 - [Usage](#)
 - [Anatomy](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

TextControl components let users enter and edit text.



Design guidelines

Usage

When to use TextControls

TextControls are best used for free text entry. If you have a set of predefined options you want users to select from, it's best to use a more constrained component, such as a SelectControl, RadioControl, CheckboxControl, or RangeControl.

Because TextControls are single-line fields, they are not suitable for collecting long responses. For those, use a text area instead.

TextControls should:

- Stand out and indicate that users can input information.
- Have clearly differentiated states (selected/unselected, active/inactive).
- Make it easy to understand the requested information and to address any errors.
- Have visible labels; placeholder text is not an acceptable replacement for a label as it vanishes when users start typing.

Anatomy



1. Label
2. Input container
3. Input text

Label text

Label text is used to inform users as to what information is requested for a text field. Every text field should have a label. Label text should be above the input field, and always visible.

Containers

Containers improve the discoverability of text fields by creating contrast between the text field and surrounding content.



Add New Tag

Do

A stroke around the container clearly indicates that users can input information.



Add New Tag



Don't

Don't use unclear visual markers to indicate a text field.

Development guidelines

Usage

Render a user interface to input the name of an additional css class.

```
import { useState } from 'react';
import { TextControl } from '@wordpress/components';

const MyTextControl = () => {
    const [ className, setClassName ] = useState( '' );

    return (
        <TextControl
            label="Additional CSS Class"
            value={ className }
```

```
        onChange={ ( value ) => setClassName( value ) }
    />
);
};
```

Props

The set of props accepted by the component will be specified below.
Props not included in this set will be applied to the input element.

label

If this property is added, a label will be generated using label property as the content.

- Type: `String`
- Required: No

hideLabelFromVision

If true, the label will only be visible to screen readers.

- Type: `Boolean`
- Required: No

help

If this property is added, a help text will be generated using help property as the content.

- Type: `String`
- Required: No

type

Type of the input element to render. Defaults to “text”.

- Type: `String`
- Required: No
- Default: “text”

value

The current value of the input.

- Type: `String` | `Number`
- Required: Yes

className

The class that will be added with “components-base-control” to the classes of the wrapper div.
If no className is passed only components-base-control is used.

- Type: `String`

- Required: No

onChange

A function that receives the value of the input.

- Type: `function`
- Required: Yes

[Related components](#)

- To offer users more constrained options for input, use SelectControl, RadioControl, CheckboxControl, or RangeControl.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: TextControl](#)”

[Previous Tabs](#) [Previous: Tabs](#)

[Next TextHighlight](#) [Next: TextHighlight](#)

TextHighlight

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [highlight: string](#)
 - [text: string](#)

[↑ Back to top](#)

Highlights occurrences of a given string within another string of text. Wraps each match with a `<mark>` tag which provides browser default styling.

[Usage](#)

Pass in the `text` and the `highlight` string to be matched against.

In the example below, the string `Gutenberg` would be highlighted twice.

```
import { TextHighlight } from '@wordpress/components';

const MyTextHighlight = () => (
  <TextHighlight
    text="Why do we like Gutenberg? Because Gutenberg is the best!"
    highlight="Gutenberg"
  />
);


```

Props

The component accepts the following props.

highlight: string

The string to search for and highlight within the `text`. Case insensitive. Multiple matches.

- Required: Yes
- Default: ''

text: string

The string of text to be tested for occurrences of then given `highlight`.

- Required: Yes
- Default: ''

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: TextHighlight](#)

[Previous TextControl](#) [Previous: TextControl](#)

[Next Text](#) [Next: Text](#)

Text

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [adjustLineHeightForInnerControls](#)

- [align](#)
- [color](#)
- [display](#)
- [ellipsis](#)
- [ellipsizeMode](#)
- [highlightCaseSensitive](#)
- [highlightEscape](#)
- [highlightSanitize](#)
- [highlightWords](#)
- [isBlock](#)
- [isDestructive](#)
- [limit](#)
- [lineHeight](#)
- [numberOfLines](#)
- [optimizeReadabilityFor](#)
- [size](#)
- [truncate](#)
- [upperCase](#)
- [variant](#)
- [weight](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Text is a core component that renders text in the library, using the library’s typography system.

Usage

Text can be used to render any text-content, like an HTML p or span.

```
import { __experimentalText as Text } from '@wordpress/components';

function Example() {
    return <Text>Code is Poetry</Text>;
}
```

Props

[adjustLineHeightForInnerControls](#)

Type: "large", "medium", "small", "xSmall"

Automatically calculate the appropriate line-height value for contents that render text and Control elements (e.g. TextInput).

```
import { __experimentalText as Text, TextInput } from '@wordpress/components';

function Example() {
    return (
        <Text adjustLineHeightForInnerControls={"small"}>
```

```
        Lorem ipsum dolor sit amet, consectetur
        <TextInput value="adipiscing elit..." />
    </Text>
);
}
```

align

Type: CSSProperties['textAlign']

Adjusts the text alignment.

```
import { __experimentalText as Text } from '@wordpress/components';

function Example() {
    return (
        <Text align="center" isBlock>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit...
        </Text>
    );
}
```

color

Type: CSSProperties['color']

Adjusts the text color.

display

Type: CSSProperties['display']

Adjusts the CSS display.

ellipsis

Type: string

The ellipsis string when truncate is set.

ellipsizeMode

Type: "auto","head","tail","middle"

Determines where to truncate. For example, we can truncate text right in the middle. To do this, we need to set ellipsizeMode to middle and a text limit.

- auto: Trims content at the end automatically without a limit.
- head: Trims content at the beginning. Requires a limit.
- middle: Trims content in the middle. Requires a limit.
- tail: Trims content at the end. Requires a limit.

[highlightCaseSensitive](#)

Type: boolean

Escape characters in `highlightWords` which are meaningful in regular expressions.

[highlightEscape](#)

Type: boolean

Determines if `highlightWords` should be case sensitive.

[highlightSanitize](#)

Type: boolean

Array of search words. String search terms are automatically cast to RegExps unless `highlightEscape` is true.

[highlightWords](#)

Type: any []

Letters or words within `Text` can be highlighted using `highlightWords`.

```
import { __experimentalText as Text } from '@wordpress/components';

function Example() {
    return (
        <Text highlightWords={ [ 'pi' ] }>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc
            neque, vulputate a diam et, luctus convallis lacus. Vestibulum
            mollis mi. Morbi id elementum massa. Suspendisse interdum auct
            ligula eget cursus. In fermentum ultricies mauris, pharetra
            fermentum erat pellentesque id.
        </Text>
    );
}
```

[isBlock](#)

Type: boolean

Sets `Text` to have `display: block`.

Note: text truncation only works when `isBlock` is `false`.

[isDestructive](#)

Type: boolean

Renders a destructive color.

limit

Type: number

Determines the max characters when `truncate` is set.

lineHeight

Type: `CSSProperties['lineHeight']`

Adjusts all text line-height based on the typography system.

numberOfLines

Type: number

Clamps the text content to the specifiec `numberOfLines`, adding the `ellipsis` at the end.

optimizeReadabilityFor

Type: `CSSProperties['color']`

The `Text` color can be adapted to a background color for optimal readability.

`optimizeReadabilityFor` can accept CSS variables, in addition to standard CSS color values (e.g. Hex, RGB, HSL, etc...).

```
import { __experimentalText as Text, View } from '@wordpress/components';

function Example() {
    const backgroundColor = 'blue';

    return (
        <View css={{ backgroundColor }}>
            <Text optimizeReadabilityFor={{ backgroundColor }}>
                Lorem ipsum dolor sit amet, consectetur adipiscing elit.
            </Text>
        </View>
    );
}
```

size

Type: `CSSProperties['fontSize'], TextSize`

Adjusts text size based on the typography system. `Text` can render a wide range of font sizes, which are automatically calculated and adapted to the typography system. The `size` value can be a system preset, a `number`, or a custom unit value (`string`) such as `30em`.

```
import { __experimentalText as Text } from '@wordpress/components';

function Example() {
    return <Text size="largeTitle">Code is Poetry</Text>;
}
```

truncate

Type: boolean

Enables text truncation. When `truncate` is set, we are able to truncate the long text in a variety of ways.

Note: text truncation won't work if the `isBlock` property is set to `true`

```
import { __experimentalText as Text } from '@wordpress/components';

function Example() {
    return (
        <Text truncate>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc
            neque, vulputate a diam et, luctus convallis lacus. Vestibulum
            mollis mi. Morbi id elementum massa. Suspendisse interdum auct
            ligula eget cursus. In fermentum ultricies mauris, pharetra
            fermentum erat pellentesque id.
        </Text>
    );
}
```

upperCase

Type: boolean

Uppercases the text content.

variant

Type: "muted"

Adjusts style variation of the text.

```
import { __experimentalText as Text } from '@wordpress/components';

function Example() {
    return <Text variant="muted">Code is Poetry</Text>;
}
```

weight

Type: CSSProperties['fontWeight'], TextWeight

Adjusts font-weight of the text.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Text”](#)

[Previous TextHighlight](#) [Previous: TextHighlight](#)

[Next TextareaControl](#) [Next: TextareaControl](#)

TextareaControl

In this article

Table of Contents

- [Design guidelines](#)
 - [Usage](#)
- [Anatomy](#)
 - [Containers](#)
 - [Label text](#)
 - [Error text](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

TextareaControls are TextControls that allow for multiple lines of text, and wrap overflow text onto a new line. They are a fixed height and scroll vertically when the cursor reaches the bottom of the field.

Write an excerpt (optional)

Write a

This c
use th
in Gut
devel

Design guidelines

Usage

When to use TextareaControl

Use TextareaControl when you need to encourage users enter an amount of text that's longer than a single line. (A bigger box can encourage people to be more verbose, where a smaller one encourages them to be succinct.)

TextareaControl should:

- Stand out from the background of the page and indicate that users can input information.
- Have clearly differentiated active/inactive states, including focus styling.
- Make it easy to understand and address any errors via clear and direct error notices.
- Make it easy to understand the requested information by using a clear and descriptive label.

When not to use TextareaControl

Do not use TextareaControl if you need to let users enter shorter answers (no longer than a single line), such as a phone number or name. In this case, you should use Text Control.



Caption

List of the most common components in Gutenberg alphabetically.

Do

Use TextareaControl to let users to enter text longer than a single line.

X

Favorite color

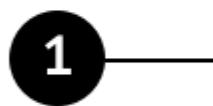
Blue

Don't

Use TextareaControl for shorter answers.

Anatomy

Write an excerpt (optional) —



1. Container
2. Label

Containers

Containers improve the discoverability of text fields by creating contrast between the text field and surrounding content.



Write an excerpt (optional)

A rectangular input field with a thin gray border, representing a container for user input.

Do

Use a stroke around the container, which clearly indicates that users can input information.



Write an excerpt (optional)

Don't

Use unclear visual markers to indicate a text field.

Label text

Label text is used to inform users as to what information is requested for a text field. Every text field should have a label. Label text should be above the input field, and always visible. Write labels in sentence capitalization.

Error text

When text input isn't accepted, an error message can display instructions on how to fix it. Error messages are displayed below the input line, replacing helper text until fixed.

Message

Please enter a message

Development guidelines

Usage

```
import { useState } from 'react';
import { TextareaControl } from '@wordpress/components';

const MyTextareaControl = () => {
    const [ text, setText ] = useState( '' );

    return (
        <TextareaControl
            label="Text"
            help="Enter some text"
            value={ text }
            onChange={ ( value ) => setText( value ) }
        />
    );
}
```

Props

The set of props accepted by the component will be specified below.

Props not included in this set will be applied to the textarea element.

`help: string | Element`

If this property is added, a help text will be generated using help property as the content.

- Required: No

`hideLabelFromVision: boolean`

If true, the label will only be visible to screen readers.

- Required: No

`label: string`

If this property is added, a label will be generated using label property as the content.

- Required: No

`onChange: (value: string) => void`

A function that receives the new value of the textarea each time it changes.

- Required: Yes

`rows: number`

The number of rows the textarea should contain.

- Required: No
- Default: 4

`value: string`

The current value of the textarea.

- Required: Yes

[Related components](#)

- For a field where users only enter one line of text, use the `TextControl` component.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: TextareaControl”](#)

[Previous Text](#) [Previous: Text](#)

[Next Theme](#) [Next: Theme](#)

Theme

In this article

Table of Contents

- [Props](#)
 - [accent: string](#)
 - [background: string](#)
- [Writing themeable components](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Theme allows defining theme variables for components in the `@wordpress/components` package.

Multiple Theme components can be nested in order to override specific theme variables.

Props

accent: string

The accent color (used by components as the primary color). If an accent color is not defined, the default fallback value is the original WP Admin main theme color.

Not all valid CSS color syntaxes are supported — in particular, keywords (like `'currentcolor'`, `'inherit'`, `'initial'`, `'revert'`, `'unset'`...) and CSS custom properties (e.g. `var(--my-custom-property)`) are *not* supported values for this property.

- Required: No

background: string

The background color. If a component explicitly has a background, it will be this color. Otherwise, this color will simply be used to determine what the foreground colors should be. The actual background color will need to be set on the component’s container element. If a background color is not defined, the default fallback value is `#fff`.

Not all valid CSS color syntaxes are supported — in particular, keywords (like `'currentcolor'`, `'inherit'`, `'initial'`, `'revert'`, `'unset'`...) and CSS custom properties (e.g. `var(--my-custom-property)`) are *not* supported values for this property.

- Required: No

[Writing themeable components](#)

If you would like your custom component to be themeable as a child of the `Theme` component, it should use these color variables. (This is a work in progress, and this list of variables may change. We do not recommend using these variables in production at this time.)

- `--wp-components-color-accent`: The accent color.
- `--wp-components-color-accent-darker-10`: A slightly darker version of the accent color.
- `--wp-components-color-accent-darker-20`: An even darker version of the accent color.
- `--wp-components-color-accent-inverted`: The foreground color when the accent color is the background, for example when placing text on the accent color.
- `--wp-components-color-background`: The background color.
- `--wp-components-color-foreground`: The foreground color, for example text.
- `--wp-components-color-foreground-inverted`: The foreground color when the foreground color is the background, for example when placing text on the foreground color.
- Grayscale:
 - `--wp-components-color-gray-100`: Used for light gray backgrounds.
 - `--wp-components-color-gray-200`: Used sparingly for light borders.
 - `--wp-components-color-gray-300`: Used for most borders.
 - `--wp-components-color-gray-400`
 - `--wp-components-color-gray-600`: Meets 3:1 UI or large text contrast against white.
 - `--wp-components-color-gray-700`: Meets 4.6:1 text contrast against white.
 - `--wp-components-color-gray-800`

First published

October 12, 2022

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: Theme”](#)

[Previous TextareaControl](#) [Previous: TextareaControl](#)
[Next ToggleControl](#) [Next: ToggleControl](#)

ToggleControl

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [label](#)
 - [help](#)
 - [checked](#)
 - [disabled](#)
 - [onChange](#)
 - [className](#)

[↑ Back to top](#)

ToggleControl is used to generate a toggle user interface.

[Usage](#)

Render a user interface to change fixed background setting.

```
import { useState } from 'react';
import { ToggleControl } from '@wordpress/components';

const MyToggleControl = () => {
    const [ hasFixedBackground, setHasFixedBackground ] = useState( false )

    return (
        <ToggleControl
            label="Fixed Background"
            help={
                hasFixedBackground
                    ? 'Has fixed background.'
                    : 'No fixed background.'
            }
            checked={ hasFixedBackground }
            onChange={ (newValue) => {
                setHasFixedBackground( newValue );
            } }
        />
    );
}
```

[Props](#)

The component accepts the following props:

label

If this property is added, a label will be generated using label property as the content.

- Type: `String`
- Required: No

help

If this property is added, a help text will be generated using help property as the content.

For controlled components the `help` prop can also be a function which will return a help text dynamically depending on the boolean `checked` parameter.

- Type: `String | Element | Function`
- Required: No

checked

If checked is true the toggle will be checked. If checked is false the toggle will be unchecked.

If no value is passed the toggle will be an uncontrolled component with unchecked initial value.

- Type: `Boolean`
- Required: No

disabled

If disabled is true the toggle will be disabled and apply the appropriate styles.

- Type: `Boolean`
- Required: No

onChange

A function that receives the checked state (boolean) as input.

- Type: `function`
- Required: No

className

The class that will be added with `components-base-control` and `components-toggle-control` to the classes of the wrapper div. If no className is passed only `components-base-control` and `components-toggle-control` are used.

- Type: `String`
- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToggleControl](#)

[Previous Theme](#) [Previous: Theme](#)

[Next ToggleGroupControlOptionBase](#) [Next: ToggleGroupControlOptionBase](#)

ToggleGroupControlOptionBase

In this article

[Table of Contents](#)

- [Props](#)
 - [children: ReactNode](#)
 - [value: string | number](#)
 - [showTooltip: boolean](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`ToggleGroupControlOptionBase` is a form component and is meant to be used as an internal, generic component for any children of [ToggleGroupControl](#).

[Props](#)

[children: ReactNode](#)

The children elements.

- Required: Yes

[value: string | number](#)

The value of the `ToggleGroupControlOptionBase`.

- Required: Yes

[showTooltip: boolean](#)

Whether to show a tooltip when hovering over the option. The tooltip will only show if a label for it is provided using the `aria-label` prop.

- Required: No

First published

March 31, 2022

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: ToggleGroupControlOptionBase”](#)

[Previous ToggleControl](#) [Previous: ToggleControl](#)

[Next ToggleGroupControlOptionIcon](#) [Next: ToggleGroupControlOptionIcon](#)

ToggleGroupControlOptionIcon

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [value: string | number](#)
 - [icon: Component](#)
 - [label: string](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`ToggleGroupControlOptionIcon` is a form component which is meant to be used as a child of [ToggleGroupControl](#) and displays an icon.

Usage

```
import {  
    __experimentalToggleGroupControl as ToggleGroupControl,  
    __experimentalToggleGroupControlOptionIcon as ToggleGroupControlOptionIcon  
} from '@wordpress/components';  
import { formatLowercase, formatUppercase } from '@wordpress/icons';  
  
function Example() {  
    return (  
        <ToggleGroupControl>  
            <ToggleGroupControlOptionIcon  
                value="uppercase"  
                icon={ formatUppercase }  
                label="Uppercase"  
            />  
            <ToggleGroupControlOptionIcon  
                value="lowercase"  
                icon={ formatLowercase }  
            />  
    );  
}
```

```
        label="Lowercase"
      />
    </ToggleGroupControl>
  );
}
```

Props

value: string | number

The value of the `ToggleGroupControl` option.

- Required: Yes

icon: Component

Icon displayed as the content of the option. Usually one of the icons from the `@wordpress/icons` package, or a custom React `<svg>` icon.

- Required: Yes

label: string

The text to accessibly label the icon option. Will also be shown in a tooltip.

- Required: Yes

First published

March 31, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: `ToggleGroupControlOptionIcon`](#)

[Previous: `ToggleGroupControlOptionBase`](#) [Previous: `ToggleGroupControlOptionBase`](#)
[Next: `ToggleGroupControlOption`](#) [Next: `ToggleGroupControlOption`](#)

ToggleGroupControlOption

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [label: string](#)

- [value: string | number](#)
- [showTooltip: boolean](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`ToggleGroupControlOption` is a form component and is meant to be used as a child of [ToggleGroupControl](#).

Usage

```
import {
  __experimentalToggleGroupControl as ToggleGroupControl,
  __experimentalToggleGroupControlOption as ToggleGroupControlOption,
} from '@wordpress/components';

function Example() {
  return (
    <ToggleGroupControl label="my label" value="vertical" isBlock>
      <ToggleGroupControlOption
        value="horizontal"
        label="Horizontal"
        showTooltip={ true }
      />
      <ToggleGroupControlOption value="vertical" label="Vertical" />
    </ToggleGroupControl>
  );
}
```

Props

label: string

Label for the option. If needed, the `aria-label` prop can be used in addition to specify a different label for assistive technologies.

- Required: Yes

value: string | number

The value of the `ToggleGroupControlOption`.

- Required: Yes

[showTooltip: boolean](#)

Whether to show a tooltip when hovering over the option. The tooltip will attempt to use the `aria-label` prop text first, then the `label` prop text if no `aria-label` prop is found.

- Required: No

First published

October 14, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToggleGroupControlOption](#)”

[Previous ToggleGroupControlIcon](#) [Previous: ToggleGroupControlOptionIcon](#)
[Next ToggleGroupControl](#) [Next: ToggleGroupControl](#)

ToggleGroupControl

In this article

[Table of Contents](#)

- [Usage](#)
- [Props](#)
 - [help: ReactNode](#)
 - [hideLabelFromVision: boolean](#)
 - [isAdaptiveWidth: boolean](#)
 - [isDeselectable: boolean](#)
 - [isBlock: boolean](#)
 - [label: string](#)
 - [onChange: \(value?: string | number \) => void](#)
 - [value: string | number](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`ToggleGroupControl` is a form component that lets users choose options represented in horizontal segments. To render options for this control use [ToggleGroupControlOption](#) component.

This component is intended for selecting a single persistent value from a set of options, similar to how a radio button group would work. If you simply want a toggle to switch between views, use a [TabPanel](#) instead.

Only use this control when you know for sure the labels of items inside won't wrap. For items with longer labels, you can consider a [SelectControl](#) or a [CustomSelectControl](#) component instead.

Usage

```
import {
    __experimentalToggleGroupControl as ToggleGroupControl,
    __experimentalToggleGroupControlOption as ToggleGroupControlOption,
} from '@wordpress/components';

function Example() {
    return (
        <ToggleGroupControl label="my label" value="vertical" isBlock>
            <ToggleGroupControlOption value="horizontal" label="Horizontal" />
            <ToggleGroupControlOption value="vertical" label="Vertical" />
        </ToggleGroupControl>
    );
}
```

Props

[help: ReactNode](#)

If this property is added, a help text will be generated using help property as the content.

- Required: No

[hideLabelFromVision: boolean](#)

If true, the label will only be visible to screen readers.

- Required: No
- Default: `false`

[isAdaptiveWidth: boolean](#)

Determines if segments should be rendered with equal widths.

- Required: No
- Default: `false`

[isDeselectable: boolean](#)

Whether an option can be deselected by clicking it again.

- Required: No
- Default: `false`

[isBlock: boolean](#)

Renders `ToggleGroupControl` as a (CSS) block element, spanning the entire width of the available space. This is the recommended style when the options are text-based and not icons.

- Required: No
- Default: `false`

[label: string](#)

Label for the form element.

- Required: Yes

[onChange: \(value?: string | number \) => void](#)

Callback when a segment is selected.

- Required: No
- Default: `() => {}`

[value: string | number](#)

The value of the `ToggleGroupControl`.

- Required: No

First published

August 20, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToggleGroupControl](#)

[Previous ToggleGroupControlOption](#) [Previous: ToggleGroupControlOption](#)
[Next ToolbarButton](#) [Next: ToolbarButton](#)

ToolbarButton

In this article

Table of Contents

- [Usage](#)
 - [Inside BlockControls](#)
- [Props](#)
- [Related components](#)

[↑ Back to top](#)

ToolbarButton can be used to add actions to a toolbar, usually inside a [Toolbar](#) or [ToolbarGroup](#) when used to create general interfaces. If you're using it to add controls to your custom block, you should consider using [BlockControls](#).

It has similar features to the [Button](#) component. Using ToolbarButton will ensure the correct styling for a button in a toolbar, and also that keyboard interactions in a toolbar are consistent with the [WAI-ARIA toolbar pattern](#).

Usage

To create general interfaces, you'll want to render ToolbarButton in a [Toolbar](#) component.

```
import { Toolbar, ToolbarButton } from '@wordpress/components';
import { edit } from '@wordpress/icons';

function MyToolbar() {
    return (
        <Toolbar label="Options">
            <ToolbarButton
                icon={ edit }
                label="Edit"
                onClick={ () => alert( 'Editing' ) }
            />
        </Toolbar>
    );
}
```

Inside BlockControls

If you're working on a custom block and you want to add controls to the block toolbar, you should use [BlockControls](#) instead. Optionally wrapping it with [ToolbarGroup](#).

```
import { BlockControls } from '@wordpress/block-editor';
import { ToolbarGroup, ToolbarButton } from '@wordpress/components';
import { edit } from '@wordpress/icons';

function Edit() {
    return (
        <BlockControls>
            <ToolbarGroup>
                <ToolbarButton
                    icon={ edit }
                    label="Edit"
                    onClick={ () => alert( 'Editing' ) }
                />
            </ToolbarGroup>
        </BlockControls>
    );
}
```

Props

This component accepts [the same API of the Button](#) component in addition to:

`containerClassName: string`

An optional additional class name to apply to the button container.

- Required: No

`subscript: string`

An optional subscript for the button.

- Required: No

Related components

- If you wish to implement a control to select options grouped as icon buttons you can use the [Toolbar](#) component, which already handles this strategy.
- The ToolbarButton may be used with other elements such as [Dropdown](#) to display options in a popover.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToolbarButton](#)

[Previous ToggleGroupControl](#) [Previous: ToggleGroupControl](#)
[Next ToolbarDropdownMenu](#) [Next: ToolbarDropdownMenu](#)

ToolbarDropdownMenu

In this article

Table of Contents

- [Usage](#)
 - [Inside BlockControls](#)
- [Props](#)

[↑ Back to top](#)

ToolbarDropdownMenu can be used to add actions to a toolbar, usually inside a [Toolbar](#) or [ToolbarGroup](#) when used to create general interfaces. If you're using it to add controls to your custom block, you should consider using [BlockControls](#).

It has similar features to the [DropdownMenu](#) component. Using ToolbarDropdownMenu will ensure that keyboard interactions in a toolbar are consistent with the [WAI-ARIA toolbar pattern](#).

Usage

To create general interfaces, you'll want to render ToolbarButton in a [Toolbar](#) component.

```
import { Toolbar, ToolbarDropdownMenu } from '@wordpress/components';
import {
    more,
    arrowLeft,
    arrowRight,
    arrowUp,
    arrowDown,
} from '@wordpress/icons';

function MyToolbar() {
    return (
        <Toolbar label="Options">
            <ToolbarDropdownMenu
                icon={ more }
                label="Select a direction"
                controls={[
                    {
                        title: 'Up',
                        icon: arrowUp,
                        onClick: () => console.log( 'up' ),
                    },
                    {
                        title: 'Right',
                        icon: arrowRight,
                        onClick: () => console.log( 'right' ),
                    },
                    {
                        title: 'Down',
                        icon: arrowDown,
                        onClick: () => console.log( 'down' ),
                    },
                    {
                        title: 'Left',
                        icon: arrowLeft,
                        onClick: () => console.log( 'left' ),
                    },
                ]}
            />
        </Toolbar>
    );
}
```

Inside BlockControls

If you're working on a custom block and you want to add controls to the block toolbar, you should use [BlockControls](#) instead.

```
import { BlockControls } from '@wordpress/block-editor';
import { Toolbar, ToolbarDropdownMenu } from '@wordpress/components';
import {
    more,
    arrowLeft,
    arrowRight,
    arrowUp,
    arrowDown,
} from '@wordpress/icons';

function Edit() {
    return (
        <BlockControls group="block">
            <ToolbarDropdownMenu
                icon={ more }
                label="Select a direction"
                controls={ [
                    {
                        title: 'Up',
                        icon: arrowUp,
                        onClick: () => console.log( 'up' ),
                    },
                    {
                        title: 'Right',
                        icon: arrowRight,
                        onClick: () => console.log( 'right' ),
                    },
                    {
                        title: 'Down',
                        icon: arrowDown,
                        onClick: () => console.log( 'down' ),
                    },
                    {
                        title: 'Left',
                        icon: arrowLeft,
                        onClick: () => console.log( 'left' ),
                    },
                ] }
            />
        </BlockControls>
    );
}
```

Props

This component accepts [the same API of the DropdownMenu](#) component.

First published

May 6, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToolbarDropdownMenu”](#)

[Previous ToolbarButton](#) [Previous: ToolbarButton](#)

[Next ToolbarGroup](#) [Next: ToolbarGroup](#)

ToolbarGroup

In this article

Table of Contents

- [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

A ToolbarGroup can be used to create subgroups of controls inside a [Toolbar](#).

Usage

```
import { Toolbar, ToolbarGroup, ToolbarButton } from '@wordpress/component'
import { paragraph, formatBold, formatItalic, link } from '@wordpress/icon'

function MyToolbar() {
    return (
        <Toolbar label="Options">
            <ToolbarGroup>
                <ToolbarButton icon={ paragraph } label="Paragraph" />
            </ToolbarGroup>
            <ToolbarGroup>
                <ToolbarButton icon={ formatBold } label="Bold" />
                <ToolbarButton icon={ formatItalic } label="Italic" />
                <ToolbarButton icon={ link } label="Link" />
            </ToolbarGroup>
        </Toolbar>
    );
}
```

Props

ToolbarGroup will pass all HTML props to the underlying element.

Related components

- ToolbarGroup may contain [ToolbarButton](#) and [ToolbarItem](#) as children.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToolbarGroup](#)

[Previous ToolbarDropdownMenu](#) [Previous: ToolbarDropdownMenu](#)
[Next ToolbarItem](#) [Next: ToolbarItem](#)

ToolbarItem

In this article

Table of Contents

- [Usage](#)
 - [as prop](#)
 - [render prop](#)
 - [Inside BlockControls](#)
- [Related components](#)

[↑ Back to top](#)

A ToolbarItem is a generic headless component that can be used to make any custom component a [Toolbar](#) item. It should be inside a [Toolbar](#) or [ToolbarGroup](#) when used to create general interfaces. If you're using it to add controls to your custom block, you should consider using [BlockControls](#).

Usage

[as prop](#)

You can use the `as` prop with a custom component or any HTML element.

```
import { Toolbar, ToolbarItem, Button } from '@wordpress/components';

function MyToolbar() {
    return (
        <Toolbar label="Options">
            <ToolbarItem as={ Button }>I am a toolbar button</ToolbarItem>
        </Toolbar>
    );
}
```

```
        <ToolbarItem as="button">I am another toolbar button</ToolbarItem>
    );
}


```

render prop

You can pass children as function to get the ToolbarItem props and pass them to another component.

```
import { Toolbar, ToolbarItem, DropdownMenu } from '@wordpress/components'
import { table } from '@wordpress/icons';

function MyToolbar() {
    return (
        <Toolbar label="Options">
            <ToolbarItem>
                { ( toolbarItemHTMLProps ) => (
                    <DropdownMenu
                        icon={ table }
                        toggleProps={ toolbarItemHTMLProps }
                        label={ 'Edit table' }
                        controls={ [] }
                    />
                ) }
            </ToolbarItem>
        </Toolbar>
    );
}
```

Inside BlockControls

If you're working on a custom block and you want to add controls to the block toolbar, you should use [BlockControls](#) instead. Optionally wrapping it with [ToolbarGroup](#).

```
import { BlockControls } from '@wordpress/block-editor';
import { ToolbarGroup, ToolbarItem, Button } from '@wordpress/components';

function Edit() {
    return (
        <BlockControls>
            <ToolbarGroup>
                <ToolbarItem as={ Button }>I am a toolbar button</ToolbarItem>
            </ToolbarGroup>
        </BlockControls>
    );
}
```

Related components

- ToolbarItem should be used inside [Toolbar](#) or [ToolbarGroup](#).
- If you want a simple toolbar button, consider using [ToolbarButton](#) instead.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToolbarItem](#)

[Previous ToolbarGroup](#) [Previous: ToolbarGroup](#)

[Next Toolbar](#) [Next: Toolbar](#)

Toolbar

In this article

[Table of Contents](#)

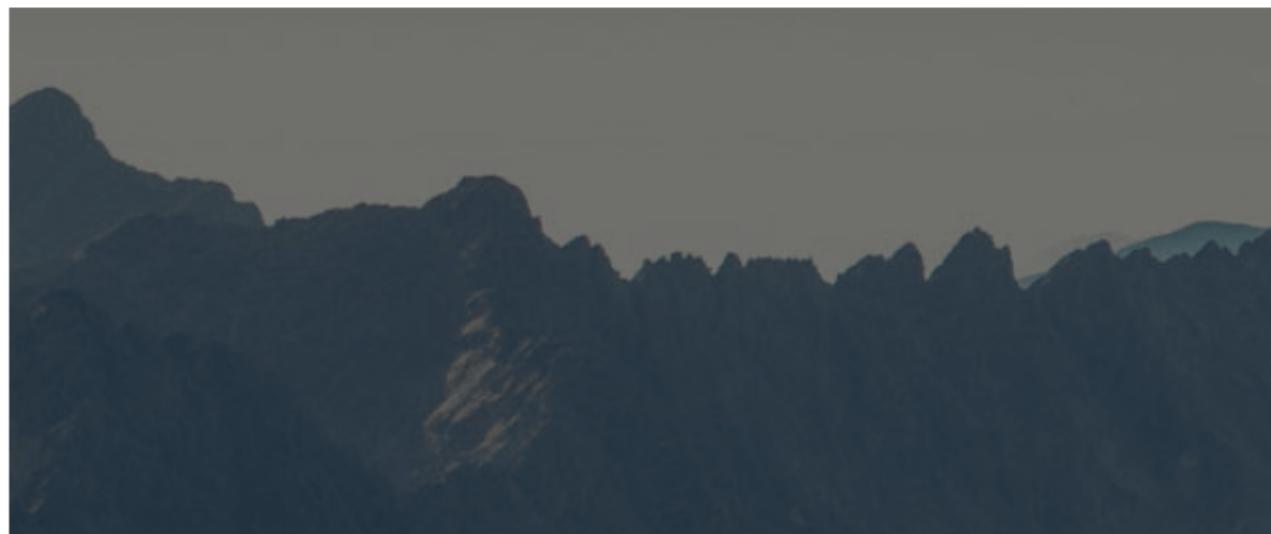
- [Design guidelines](#)
 - [Usage](#)
 - [Best practices](#)
 - [States](#)
- [Development guidelines](#)
 - [Usage](#)
 - [Props](#)
- [Related components](#)

[↑ Back to top](#)

Toolbar can be used to group related options. To emphasize groups of related icon buttons, a toolbar should share a common container.

Welcome

Editor



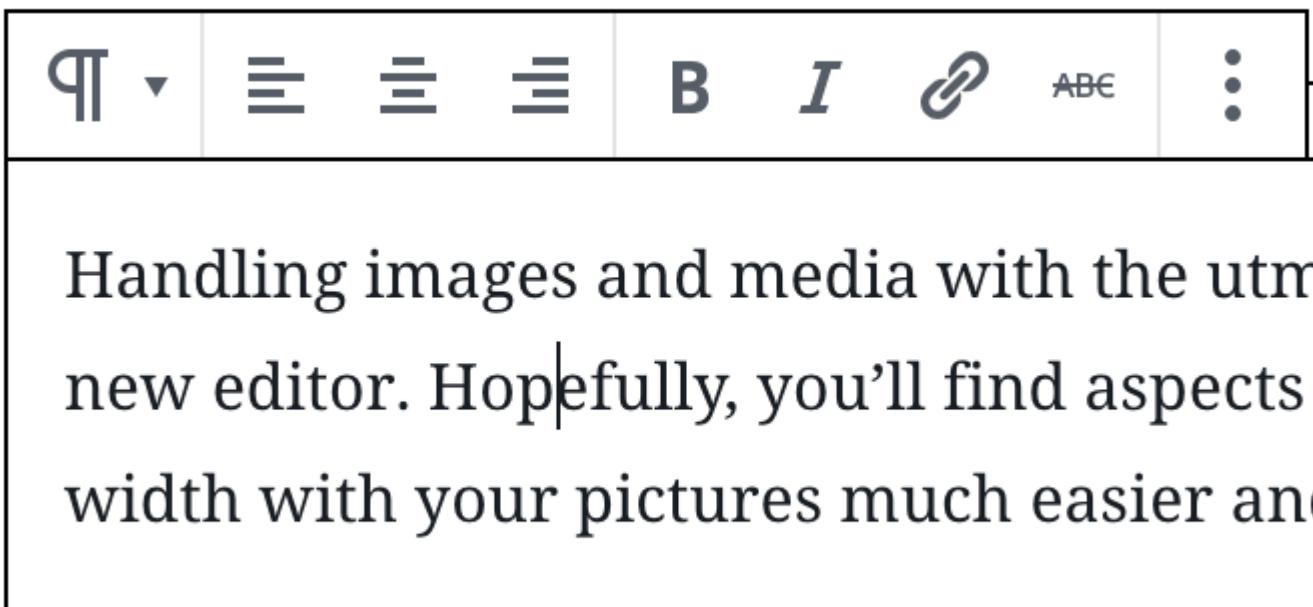
Design guidelines

Usage

Best practices

Toolbars should:

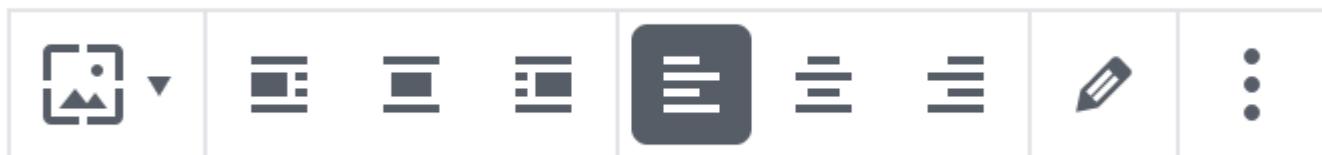
- Clearly communicate that clicking or tapping will trigger an action.
- Use established colors appropriately. For example, only use red for actions that are difficult or impossible to undo.
- When used with a block, have a consistent location above the block. Otherwise, have a consistent location in the interface.



States

Active and available toolbars

A toolbar's state makes it clear which icon button is active. Hover and focus states express the available selection options for icon buttons in a toolbar.



Disabled toolbars

Toolbars that cannot be selected can either be given a disabled state, or be hidden.

Development guidelines

Usage

```
import { Toolbar, ToolbarButton } from '@wordpress/components';
import { formatBold, formatItalic, link } from '@wordpress/icons';
```

```
function MyToolbar() {
  return (
    <Toolbar label="Options">
      <ToolbarButton icon={ formatBold } label="Bold" />
      <ToolbarButton icon={ formatItalic } label="Italic" />
      <ToolbarButton icon={ link } label="Link" />
    </Toolbar>
  );
}
```

Props

Toolbar will pass all HTML props to the underlying element. Additionally, you can pass the custom props specified below.

className: string

Class to set on the container div.

- Required: No

label: string

An accessible label for the toolbar.

- Required: Yes

variant: 'unstyled' | undefined

Specifies the toolbar's style.

Leave undefined for the default style. Or 'unstyled' which removes the border from the toolbar, but keeps the default popover style.

- Required: No
- Default: undefined

Related components

- Toolbar may contain [ToolbarGroup](#), [ToolbarButton](#) and [ToolbarItem](#) as children.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Toolbar](#)

ToolsPanelHeader

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [dropdownMenuProps: {}](#)
 - [headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’](#)
 - [label: string](#)
 - [resetAll: \(\) => void](#)
 - [toggleItem: \(label: string \) => void](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

This component renders a tools panel’s header including a menu.

[Usage](#)

This component is generated automatically by its parent `ToolsPanel`.

In general, this should not be used directly.

[Props](#)

[dropdownMenuProps: {}](#)

The dropdown menu props to configure the panel’s `DropdownMenu`.

- Type: `DropdownMenuProps`
- Required: No

[headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’](#)

The heading level of the panel’s header.

- Required: No
- Default: 2

[label: string](#)

Text to be displayed within the panel header. It is also passed along as the `label` for the panel header's `DropdownMenu`.

- Required: Yes

[resetAll: \(\) => void](#)

The `resetAll` prop provides the callback to execute when the “Reset all” menu item is selected. Its purpose is to facilitate resetting any control values for items contained within this header’s panel.

- Required: Yes

[toggleItem: \(label: string \) => void](#)

This is executed when an individual control’s menu item is toggled. It will update the panel’s menu item state and call the panel item’s `onSelect` or `onDeselect` callbacks as appropriate.

- Required: Yes

First published

August 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToolsPanelHeader](#)

[Previous Toolbar](#) [Previous: Toolbar](#)

[Next ToolsPanelItem](#) [Next: ToolsPanelItem](#)

ToolsPanelItem

In this article

Table of Contents

- [Usage](#)
- [Props](#)

- [hasValue: \(\) => boolean](#)
- [isShownByDefault: boolean](#)
- [label: string](#)
- [onDeselect: \(\) => void](#)
- [onSelect: \(\) => void](#)
- [panelId: string | null](#)

- [`resetAllFilter: \(attributes?: any \) => any`](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

This component acts as a wrapper and controls the display of items to be contained within a ToolsPanel. An item is displayed if it is flagged as a default control or the corresponding panel menu item, provided via context, is toggled on for this item.

Usage

See [`tools-panel/README.md#usage`](#) for how to use ToolsPanelItem.

Props

hasValue: () => boolean

This is called when building the ToolsPanel menu to determine the item’s initial checked state.

- Required: Yes

isShownByDefault: boolean

This prop identifies the current item as being displayed by default. This means it will show regardless of whether it has a value set or is toggled on in the panel’s menu.

- Required: No
- Default: `false`

label: string

The supplied label is dual purpose.

It is used as:

1. the human-readable label for the panel’s dropdown menu
2. a key to locate the corresponding item in the panel’s menu context to determine if the panel item should be displayed.

A panel item’s `label` should be unique among all items within a single panel.

- Required: Yes

[onDeselect: \(\) => void](#)

Called when this item is deselected in the ToolsPanel menu. This is normally used to reset the panel item control's value.

- Required: No

[onSelect: \(\) => void](#)

A callback to take action when this item is selected in the ToolsPanel menu.

- Required: No

[panelId: string | null](#)

Panel items will ensure they are only registering with their intended panel by comparing the panelId props set on both the item and the panel itself, or if the panelId is explicitly null. This allows items to be injected from a shared source.

- Required: No

[resetAllFilter: \(attributes?: any \) => any](#)

A ToolsPanel will collect each item's resetAllFilter and pass an array of these functions through to the panel's resetAll callback. They can then be iterated over to perform additional tasks.

- Required: No
- Default: () => {}

First published

August 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToolsPanelItem”](#)

[Previous ToolsPanelHeader](#) [Previous: ToolsPanelHeader](#)

[Next ToolsPanel](#) [Next: ToolsPanel](#)

ToolsPanel

In this article

Table of Contents

- [Development guidelines](#)
 - [ToolsPanel Layout](#)
- [Usage](#)
- [Props](#)
 - [hasInnerWrapper: boolean](#)
 - [dropdownMenuProps: {}](#)
 - [headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | '1' | '2' | '3' | '4' | '5' | '6'](#)
 - [label: string](#)
 - [panelId: string | null](#)
 - [resetAll: \(filters?: ResetAllFilter\[\] \) => void](#)
 - [shouldRenderPlaceholderItems: boolean](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

These panels provide progressive discovery options for their children. For example the controls provided via block supports.

[Development guidelines](#)

The ToolsPanel creates a container with a header including a dropdown menu. The menu is generated automatically from the panel’s children matching the ToolsPanelItem component type.

Each menu item allows for the display of the corresponding child to be toggled on or off. The control’s onSelect and onDeselect callbacks are fired allowing for greater control over the child e.g. resetting block attributes when a block support control is toggled off.

Whether a child control is initially displayed or not is dependent upon if there has previously been a value set or the child has been flagged as displaying by default through the isShownByDefault prop. Determining whether a child has a value is done via the hasValue function provided through the child’s props.

Components that are not wrapped within a ToolsPanelItem are still rendered however they will not be represented within, or controlled by, the ToolsPanel menu. An example scenario that benefits from this could be displaying introduction or help text within a panel.

ToolsPanel Layout

The ToolsPanel has a two-column grid layout. By default, ToolsPanelItem components within the panel are styled to span both columns as this fits the majority of use-cases. Most non-control elements, such as help text, will be rendered as children of the related control's ToolsPanelItem and not require additional styling.

Suppose an element is related to multiple controls (e.g. a contrast checker), or the panel itself (e.g. a panel description). In that case, these will be rendered into the panel without a wrapping ToolsPanelItem. They'll then only span a single column by default. If this is undesirable, those elements will likely need a small style tweak, e.g. `grid-column: 1 / -1;`

The usage example below will illustrate a non-ToolsPanelItem description paragraph, controls that should display in a single row, and others spanning both columns.

Usage

```
/**  
 * External dependencies  
 */  
import styled from '@emotion/styled';  
  
/**  
 * WordPress dependencies  
 */  
import {  
    __experimentalBoxControl as BoxControl,  
    __experimentalToolsPanel as ToolsPanel,  
    __experimentalToolsPanelItem as ToolsPanelItem,  
    __experimentalUnitControl as UnitControl,  
} from '@wordpress/components';  
import { __ } from '@wordpress/i18n';  
  
const PanelDescription = styled.div`  
    grid-column: span 2;  
`;  
  
const SingleColumnItem = styled( ToolsPanelItem )`  
    grid-column: span 1;  
`;  
  
export function DimensionPanel() {  
    const [ height, setHeight ] = useState();  
    const [ width, setWidth ] = useState();  
    const [ padding, setPadding ] = useState();  
    const [ margin, setMargin ] = useState();  
  
    const resetAll = () => {  
        setHeight( undefined );  
        setWidth( undefined );  
    };  
}
```

```

        setPadding( undefined );
        setMargin( undefined );
    };

    return (
        <ToolsPanel label={ __( 'Dimensions' ) } resetAll={ resetAll }>
            <PanelDescription>
                Select dimensions or spacing related settings from the
                menu for additional controls.
            </PanelDescription>
            <SingleColumnItem
                hasValue={ () => !! height }
                label={ __( 'Height' ) }
                onDeselect={ () => setHeight( undefined ) }
                isShownByDefault
            >
                <UnitControl
                    label={ __( 'Height' ) }
                    onChange={ setHeight }
                    value={ height }
                />
            </SingleColumnItem>
            <SingleColumnItem
                hasValue={ () => !! width }
                label={ __( 'Width' ) }
                onDeselect={ () => setWidth( undefined ) }
                isShownByDefault
            >
                <UnitControl
                    label={ __( 'Width' ) }
                    onChange={ setWidth }
                    value={ width }
                />
            </SingleColumnItem>
            <ToolsPanelItem
                hasValue={ () => !! padding }
                label={ __( 'Padding' ) }
                onDeselect={ () => setPadding( undefined ) }
            >
                <BoxControl
                    label={ __( 'Padding' ) }
                    onChange={ setPadding }
                    values={ padding }
                    allowReset={ false }
                />
            </ToolsPanelItem>
            <ToolsPanelItem
                hasValue={ () => !! margin }
                label={ __( 'Margin' ) }
                onDeselect={ () => setMargin( undefined ) }
            >
                <BoxControl
                    label={ __( 'Margin' ) }
                    onChange={ setMargin }
                />
            </ToolsPanelItem>
        </ToolsPanel>
    );
}

```

```
        values={ margin }
        allowReset={ false }
    />
</ToolsPanelItem>
</ToolsPanel>
);
}
```

Props

hasInnerWrapper: boolean

Flags that the items in this ToolsPanel will be contained within an inner wrapper element allowing the panel to lay them out accordingly.

- Required: No
- Default: `false`

dropdownMenuProps: {}

The popover props to configure panel's DropdownMenu.

- Type: `DropdownMenuProps`
- Required: No

headingLevel: 1 | 2 | 3 | 4 | 5 | 6 | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’

The heading level of the panel's header.

- Required: No
- Default: 2

label: string

Text to be displayed within the panel's header and as the `aria-label` for the panel's dropdown menu.

- Required: Yes

panelId: string | null

If a `panelId` is set, it is passed through the `ToolsPanelContext` and used to restrict panel items. When a `panelId` is set, items can only register themselves if the `panelId` is explicitly `null` or the item's `panelId` matches exactly.

- Required: No

[resetAll: \(filters?: ResetAllFilter\[\] \) => void](#)

A function to call when the `Reset all` menu option is selected. As an argument, it receives an array containing the `resetAllFilter` callbacks of all the valid registered `ToolsPanelItems`.

- Required: Yes

[shouldRenderPlaceholderItems: boolean](#)

Advises the `ToolsPanel` that all of its `ToolsPanelItem` children should render placeholder content (instead of `null`) when they are toggled off and hidden.

Note that placeholder items won't apply the `className` that would be normally applied to a visible `ToolsPanelItem` via the `className` prop.

- Required: No
- Default: `false`

First published

August 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ToolsPanel”](#)

[Previous ToolsPanelItem](#) [Previous: ToolsPanelItem](#)

[Next Tooltip](#) [Next: Tooltip](#)

Tooltip

In this article

[Table of Contents](#)

- [Usage](#)
 - [Nested tooltips](#)
- [Props](#)

[↑ Back to top](#)

Tooltip is a React component to render floating help text relative to a node when it receives focus or it is hovered upon by a mouse. If the tooltip exceeds the bounds of the page in the direction it opens, its position will be flipped automatically.

Usage

Render a Tooltip, passing as a child the element to which it should anchor:

```
import { Tooltip } from '@wordpress/components';

const MyTooltip = () => (
    <Tooltip text="More information">
        <div>Hover for more information</div>
    </Tooltip>
);
```

Nested tooltips

In case one or more `Tooltip` components are rendered inside another `Tooltip` component, only the tooltip associated to the outermost `Tooltip` component will be rendered in the browser and shown to the user appropriately. The rest of the nested `Tooltip` components will simply no-op and pass-through their anchor.

Props

The component accepts the following props:

`children: React.ReactNode`

The element to which the tooltip should anchor.

NOTE: Accepts only one child element.

- Required: Yes

`delay: number`

The amount of time in milliseconds to wait before showing the tooltip.

- Required: No
- Default: 700

`hideOnClick: boolean`

Option to hide the tooltip when the anchor is clicked.

- Required: No
- Default: true

```
placement: 'top' | 'top-start' | 'top-end' | 'right' | 'right-start' | 'right-end' | 'bottom' | 'bottom-start' | 'bottom-end' | 'left' | 'left-start' | 'left-end'
```

Used to specify the tooltip's placement with respect to its anchor.

- Required: No
- Default: 'bottom'

`position: string`

Note: use the `placement` prop instead when possible.

Legacy way to specify the popover's position with respect to its anchor. Specify y- and x-axis as a space-separated string. Supports 'top', 'middle', 'bottom' y axis, and 'left', 'center', 'right' x axis.

- Required: No
- Default: 'bottom'

`shortcut: string | object`

An option for adding accessible keyboard shortcuts.

If shortcut is a string, it is expecting the display text. If shortcut is an object, it will accept the properties of `display: string` and `ariaLabel: string`.

- Required: No

`text: string`

The text shown in the tooltip when anchor element is focused or hovered.

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Tooltip](#)

[Previous ToolsPanel](#) [Previous: ToolsPanel](#)
[Next TreeGrid](#) [Next: TreeGrid](#)

TreeGrid

In this article

Table of Contents

- [Development guidelines](#)
 - [Usage](#)
 - [Sub-Components](#)
 - [TreeGridCell](#)
- [Related components](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

[Development guidelines](#)

TreeGrid, TreeGridRow, and TreeGridCell are components used to create a tree hierarchy. They’re not visually styled components, but instead help with adding keyboard navigation and roving tabindex behaviors to tree grid structures.

A tree grid is a hierarchical 2 dimensional UI component, for example it could be used to implement a file system browser.

A tree grid allows the user to navigate using arrow keys. Up/down to navigate vertically across rows, and left/right to navigate horizontally between focusables in a row.

For more information on a tree grid, see the following links:

- <https://www.w3.org/TR/wai-aria-practices/examples/treegrid/treegrid-1.html>

[Usage](#)

The TreeGrid renders both a table and tbody element, and is intended to be used with TreeGridRow(tr) and TreeGridCell(td) to build out a grid.

```
function TreeMenu() {  
    return (  
        <TreeGrid>  
            <TreeGridRow level={ 1 } positionInSet={ 1 } setSize={ 2 }>  
                <TreeGridCell>  
                    { ( props ) => (  
                        <Button onClick={ onSelect } { ...props }>Select</  
                        > ) }  
                </TreeGridCell>  
                <TreeGridCell>  
                    { ( props ) => (  
                        <Button onClick={ onMove } { ...props }>Move</Butt  
                        > ) }  
                </TreeGridCell>  
            </TreeGridRow>  
        </TreeGrid>  
    )  
}
```

```

        )
      </TreeGridCell>
    </TreeGridRow>
    <TreeGridRow level={ 1 } positionInSet={ 2 } setSize={ 2 }>
      <TreeGridCell>
        { ( props ) => (
          <Button onClick={ onSelect } { ...props }>Select</
        ) }
      </TreeGridCell>
      <TreeGridCell>
        { ( props ) => (
          <Button onClick={ onMove } { ...props }>Move</Butt
        ) }
      </TreeGridCell>
    </TreeGridRow>
    <TreeGridRow level={ 2 } positionInSet={ 1 } setSize={ 1 }>
      <TreeGridCell>
        { ( props ) => (
          <Button onClick={ onSelect } { ...props }>Select</
        ) }
      </TreeGridCell>
      <TreeGridCell>
        { ( props ) => (
          <Button onClick={ onMove } { ...props }>Move</Butt
        ) }
      </TreeGridCell>
    </TreeGridRow>
  </TreeGrid>
);
}

```

Sub-Components

TreeGrid

Props

Aside from the documented callback functions, any props specified will be passed to the `table` element rendered by `TreeGrid`.

`TreeGrid` should always have children.

`onFocusRow: (event: KeyboardEvent, startRow: HTMLElement, destinationRow: HTMLElement) => void`

Callback that fires when focus is shifted from one row to another via the Up and Down keys. Callback is also fired on Home and End keys which move focus from the beginning row to the end row.

The callback is passed the event, the start row element that the focus was on originally, and the destination row element after the focus has moved.

- Required: No

```
onCollapseRow: ( row: HTMLElement ) => void
```

A callback that passes in the row element to be collapsed.

- Required: No

```
onExpandRow: ( row: HTMLElement ) => void
```

A callback that passes in the row element to be expanded.

- Required: No

TreeGridRow

Props

Additional props other than those specified below will be passed to the `tr` element rendered by `TreeGridRow`, so for example, it is possible to also set a `className` on a row.

`level: number`

An integer value designating the level in the hierarchical tree structure. Counting starts at 1. A value of 1 indicates the root level of the structure.

- Required: Yes

`positionInSet: number`

An integer value that represents the position in the set. A set is the count of elements at a specific level. Counting starts at 1.

- Required: Yes

`setSize: number`

An integer value that represents the total number of items in the set, at this specific level of the hierarchy.

- Required: Yes

`isExpanded: boolean`

An optional value that designates whether a row is expanded or collapsed. Currently this value only sets the correct `aria-expanded` property on a row, it has no other built-in behavior.

If there is a need to implement `aria-expanded` elsewhere in the row, cell, or element within a cell, you may pass `isExpanded={ undefined }`. In order for keyboard navigation to continue working, add the `data-expanded` attribute with either `true/false`. This allows the `TreeGrid` component to still manage keyboard interactions while allowing the `aria-expanded` attribute to be placed elsewhere. See the example below.

- Required: No

```
function TreeMenu() {
  return (
    <TreeGrid>
      <TreeGridRow level={ 1 } positionInSet={ 1 } setSize={ 2 } isE...
        <TreeGridCell>
          { ( props ) => (
            <Button aria-expanded="false" onClick={ onSelect } ...
          ) }
        </TreeGridCell>
      </TreeGridRow>
    </TreeGrid>
  );
}
```

[TreeGridCell](#)

Props

`TreeGridCell` accepts no specific props. Any props specified will be passed to the `td` element rendered by `TreeGridCell`.

children as a function

`TreeGridCell` renders children using a function:

```
<TreeGridCell>
  { ( props ) => (
    <Button onClick={ doSomething } { ...props }>
      Do something
    </Button>
  ) }
</TreeGridCell>
```

Props passed as an argument to the render prop must be passed to the child focusable component/element within the cell. If a component is used, it must correctly handle the `onFocus`, `tabIndex`, and `ref` props, passing these to the element it renders. These props are used to handle the roving tabindex functionality of the tree grid.

[Related components](#)

- This component implements `RovingTabIndex`.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: TreeGrid](#)

TreeSelect

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [label](#)
 - [noOptionLabel](#)
 - [onChange](#)
 - [selectedId](#)
 - [tree](#)

[↑ Back to top](#)

TreeSelect component is used to generate select input fields.

Usage

Render a user interface to select the parent page in a hierarchy of pages:

```
import { useState } from 'react';
import { TreeSelect } from '@wordpress/components';

const MyTreeSelect = () => {
    const [ page, setPage ] = useState( 'p21' );

    return (
        <TreeSelect
            label="Parent page"
            noOptionLabel="No parent page"
            onChange={ ( newPage ) => setPage( newPage ) }
            selectedId={ page }
            tree={ [
                {
                    name: 'Page 1',
                    id: 'p1',
                    children: [
                        { name: 'Descend 1 of page 1', id: 'p11' },
                        { name: 'Descend 2 of page 1', id: 'p12' },
                    ],
                },
                {
                    name: 'Page 2',
                    id: 'p2',
                    children: [

```

```

        {
          name: 'Descend 1 of page 2',
          id: 'p21',
          children: [
            {
              name: 'Descend 1 of Descend 1 of page 2',
              id: 'p211',
            },
          ],
        },
      ],
    },
  ],
);
}

```

Props

The set of props accepted by the component will be specified below.
 Props not included in this set will be applied to the SelectControl component being used.

label

If this property is added, a label will be generated using label property as the content.

- Type: `String`
- Required: No

noOptionLabel

If this property is added, an option will be added with this label to represent empty selection.

- Type: `String`
- Required: No

onChange

A function that receives the id of the new node element that is being selected.

- Type: `function`
- Required: Yes

selectedId

The id of the currently selected node.

- Type: `string | string[]`
- Required: No

tree

An array containing the tree objects with the possible nodes the user can select.

- Type: Object []
- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: TreeSelect”](#)

[Previous TreeGrid](#) [Previous: TreeGrid](#)

[Next Truncate](#) [Next: Truncate](#)

Truncate

In this article

Table of Contents

- [Usage](#)
- [Props](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Truncate is a typography primitive that trims text content. For almost all cases, it is recommended that Text, Heading, or Subheading is used to render text content. However, Truncate is available for custom implementations.

Usage

```
import { __experimentalTruncate as Truncate } from '@wordpress/components'

function Example() {
    return (
        <Truncate>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc
            neque, vulputate a diam et, luctus convallis lacus. Vestibulum
            mollis mi. Morbi id elementum massa.
        </Truncate>
    )
}
```

```
) ;  
}
```

Props

`children: ReactNode`

The `children` elements.

Note: text truncation will be attempted only if the `children` are either of type `string` or `number`. In any other scenarios, the component will not attempt to truncate the text, and will pass through the `children`.

- Required: Yes

`ellipsis: string`

The ellipsis string when truncating the text by the `limit` prop's value.

- Required: No
- Default: `...`

`ellipsizeMode: 'auto' | 'head' | 'tail' | 'middle' | 'none'`

Determines where to truncate. For example, we can truncate text right in the middle. To do this, we need to set `ellipsizeMode` to `middle` and a text `limit`.

- `auto`: Trims content at the end automatically without a `limit`.
 - `head`: Trims content at the beginning. Requires a `limit`.
 - `middle`: Trims content in the middle. Requires a `limit`.
 - `tail`: Trims content at the end. Requires a `limit`.
- Required: No
 - Default: `auto`

`limit: number`

Determines the max number of characters to be displayed before the rest of the text gets truncated. Requires `ellipsizeMode` to assume values different from `auto` and `none`.

- Required: No
- Default: `0`

`numberOfLines: number`

Clamps the text content to the specified `numberOfLines`, adding an ellipsis at the end. Note: this feature ignores the value of the `ellipsis` prop and always displays the default `...` ellipsis.

- Required: No
- Default: `0`

```
import { __experimentalTruncate as Truncate } from '@wordpress/components'

function Example() {
    return (
        <Truncate numberOfLines={ 2 }>
            Where the north wind meets the sea, there's a river full of me
            Sleep, my darling, safe and sound, for in this river all is fo
            In her waters, deep and true, lay the answers and a path for y
            Dive down deep into her sound, but not too far or you'll be dr
        </Truncate>
    );
}
```

First published

April 30, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Truncate”](#)

[Previous TreeSelect](#) [Previous: TreeSelect](#)

[Next UnitControl](#) [Next: UnitControl](#)

UnitControl

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [disableUnits: boolean](#)
 - [isPressEnterToChange: boolean](#)
 - [isResetValueOnUnitChange: boolean](#)
 - [isUnitSelectTabbable: boolean](#)
 - [label: string](#)
 - [labelPosition: string](#)
 - [onBlur: FocusEventHandler< HTMLInputElement | HTMLSelectElement >](#)
 - [onFocus: FocusEventHandler< HTMLInputElement | HTMLSelectElement >](#)
 - [onChange: UnitControlOnChangeCallback](#)
 - [onUnitChange: UnitControlOnChangeCallback](#)
 - [size: string](#)
 - [unit: string](#)
 - [units: WPUnitControlUnit\[\]](#)
 - [value: number | string](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

`UnitControl` allows the user to set a numeric quantity as well as a unit (e.g. `px`).

Usage

```
import { useState } from 'react';
import { __experimentalUnitControl as UnitControl } from '@wordpress/compo

const Example = () => {
  const [ value, setValue ] = useState( '10px' );

  return <UnitControl onChange={ setValue } value={ value } />;
};
```

Props

disableUnits: boolean

If true, the unit `<select>` is hidden.

- Required: No
- Default: `false`

isPressEnterToChange: boolean

If true, the ENTER key press is required in order to trigger an `onChange`. If enabled, a change is also triggered when tabbing away (`onBlur`).

- Required: No
- Default: `false`

isResetValueOnUnitChange: boolean

If true, and the selected unit provides a default value, this value is set when changing units.

- Required: No
- Default: `false`

isUnitSelectTabbable: boolean

Determines if the unit `<select>` is tabbable.

- Required: No
- Default: `true`

label: string

If this property is added, a label will be generated using `label` property as the content.

- Required: No

labelPosition: string

The position of the label (top, side, bottom, or edge).

- Required: No

onBlur: FocusEventHandler< HTMLInputElement | HTMLSelectElement >

Callback invoked when either the quantity or unit inputs fire the blur event.

- Required: No

onFocus: FocusEventHandler< HTMLInputElement | HTMLSelectElement >

Callback invoked when either the quantity or unit inputs fire the focus event.

- Required: No

onChange: UnitControlOnChangeCallback

Callback when the value changes.

- Required: No

onUnitChange: UnitControlOnChangeCallback

Callback when the unit changes.

- Required: No

size: string

Adjusts the size of the input.

Sizes include: default, small

- Required: No
- Default: default

unit: string

Deprecated: Current unit value.

Instead, provide a unit with a value through the value prop.

Example:

```
<UnitControl value="50%" />
```

- Required: No

[units: WPUnitControlUnit\[\]](#)

Collection of available units.

- Required: No

Example:

```
import { useState } from 'react';
import { __experimentalUnitControl as UnitControl } from '@wordpress/compo

const Example = () => {
    const [ value, setValue ] = useState( '10px' );

    const units = [
        { value: 'px', label: 'px', default: 0 },
        { value: '%', label: '%', default: 10 },
        { value: 'em', label: 'em', default: 0 },
    ];

    return (
        <UnitControl onChange={ setValue } value={ value } units={ units } />
    );
}
```

A `default` value (in the example above, `10` for `%`), if defined, is set as the new `value` when a unit changes. This is helpful in scenarios where changing a unit may cause drastic results, such as changing from `px` to `vh`.

[value: number | string](#)

Current value. If passed as a string, the current unit will be inferred from this value. For example, a `value` of `50%` will set the current unit to `%`.

Example:

```
<UnitControl value="50%" />
```

- Required: No

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: UnitControl](#)

[Previous Truncate](#) [Previous: Truncate](#)

[Next VStack](#) [Next: VStack](#)

VStack

In this article

Table of Contents

- [Usage](#)
- [Props](#)
- [Spacer](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

VStack (or Vertical Stack) is a layout component that arranges child elements in a vertical line.

[Usage](#)

VStack can render anything inside.

```
import {
    __experimentalText as Text,
    __experimentalVStack as VStack,
} from '@wordpress/components';

function Example() {
    return (
        <VStack>
            <Text>Code</Text>
            <Text>is</Text>
            <Text>Poetry</Text>
        </VStack>
    );
}
```

[Props](#)

`alignment: HStackAlignment | CSSProperties['alignItems']`

Determines how the child elements are aligned.

- `top`: Aligns content to the top.
- `topLeft`: Aligns content to the top/left.
- `topRight`: Aligns content to the top/right.
- `left`: Aligns content to the left.
- `center`: Aligns content to the center.
- `right`: Aligns content to the right.
- `bottom`: Aligns content to the bottom.

- **bottomLeft**: Aligns content to the bottom/left.
- **bottomRight**: Aligns content to the bottom/right.
- **edge**: Justifies content to be evenly spread out up to the main axis edges of the container.
- **stretch**: Stretches content to the cross axis edges of the container.

`direction: FlexDirection`

The direction flow of the children content can be adjusted with `direction`. `column` will align children vertically and `row` will align children horizontally.

`expanded: boolean`

Expands to the maximum available width (if horizontal) or height (if vertical).

`justify: CSSProperties['justifyContent']`

Horizontally aligns content if the `direction` is `row`, or vertically aligns content if the `direction` is `column`.

In the example below, `flex-start` will align the children content to the left.

`spacing: CSSProperties['width']`

The amount of space between each child element. Spacing in between each child can be adjusted by using `spacing`.

The value of `spacing` works as a multiplier to the library's grid system (base of 4px).

`wrap: boolean`

Determines if children should wrap.

Spacer

When a `Spacer` is used within an `VStack`, the `Spacer` adaptively expands to take up the remaining space.

```
import {
  __experimentalSpacer as Spacer,
  __experimentalText as Text,
  __experimentalVStack as VStack,
} from '@wordpress/components';

function Example() {
  return (
    <VStack>
      <Text>Code</Text>
      <Spacer>
        <Text>is</Text>
      </Spacer>
      <Text>Poetry</Text>
    </VStack>
  );
}
```

`Spacer` can also be used in-between items to push them apart.

```
import {
  __experimentalSpacer as Spacer,
  __experimentalText as Text,
  __experimentalVStack as VStack,
} from '@wordpress/components';

function Example() {
  return (
    <VStack>
      <Text>Code</Text>
      <Spacer />
      <Text>is</Text>
      <Text>Poetry</Text>
    </VStack>
  );
}
```

First published

May 6, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: `VStack`](#)

[Previous UnitControl](#) [Previous: UnitControl](#)
[Next VisuallyHidden](#) [Next: VisuallyHidden](#)

VisuallyHidden

In this article

[Table of Contents](#)

- [Usage](#)
- [Best practices](#)

[↑ Back to top](#)

`VisuallyHidden` is a component used to render text intended to be visually hidden, but will show for alternate devices, for example a screen reader.

Usage

```
import { VisuallyHidden } from '@wordpress/components';

function Example() {
    return (
        <VisuallyHidden>
            <label>Code is Poetry</label>
        </VisuallyHidden>
    );
}
```

Best practices

The element that `VisuallyHidden` renders has the style `position: absolute`. When using this component be careful of the [stacking context](#). Even though `VisuallyHidden` isn't visible, it can still affect layout. An example of this is that `VisuallyHidden` may ignore `overflow` styles of ancestor elements because it instead adopts the `overflow` of its stacking context. One known side-effect can be an unexpected scrollbar appearing. To fix this kind of issue, introduce a stacking context on a more immediate parent of `VisuallyHidden`. Adding `position: relative` is often an easy way to do this.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: VisuallyHidden”](#)

[Previous VStack](#) [Previous: VStack](#)

[Next ZStack](#) [Next: ZStack](#)

ZStack

In this article

Table of Contents

- [Usage](#)
- [Props](#)
 - [isLayered: boolean](#)
 - [isReversed: boolean](#)
 - [offset: number](#)
 - [children: ReactNode](#)

[↑ Back to top](#)

This feature is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Usage

ZStack allows you to stack things along the Z-axis.

```
import { __experimentalZStack as ZStack } from '@wordpress/components';

function Example() {
    return (
        <ZStack offset={ 20 } isLayered>
            <ExampleImage />
            <ExampleImage />
            <ExampleImage />
        </ZStack>
    );
}
```

Props

isLayered: boolean

Layers children elements on top of each other (first: highest z-index, last: lowest z-index).

- Required: No
- Default: true

isReversed: boolean

Reverse the layer ordering (first: lowest z-index, last: highest z-index).

- Required: No
- Default: false

offset: number

The amount of space between each child element. Its value is automatically inverted (i.e. from positive to negative, and viceversa) when switching from LTR to RTL.

- Required: No
- Default: 0

children: ReactNode

The children to stack.

- Required: Yes

First published

May 21, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: ZStack”](#)

[Previous VisuallyHidden](#) [Previous: VisuallyHidden](#)

[Next Package Reference](#) [Next: Package Reference](#)

Package Reference

In this article

Table of Contents

- [Using the packages via WordPress global](#)
- [Using the packages via npm](#)
- [Testing JavaScript code from a specific major WordPress version](#)

[↑ Back to top](#)

WordPress exposes a list of JavaScript packages and tools for WordPress development.

Using the packages via WordPress global

JavaScript packages are available as a registered script in WordPress and can be accessed using the `wp` global variable.

If you wanted to use the `PlainText` component from the block editor module, first you would specify `wp-block-editor` as a dependency when you enqueue your script:

```
wp_enqueue_script(
    'my-custom-block',
    plugins_url( $block_path, __FILE__ ),
    array( 'react', 'wp-blocks', 'wp-block-editor', 'wp-i18n' )
);
```

After the dependency is declared, you can access the module in your JavaScript code using the global `wp` like so:

```
const { PlainText } = wp.blockEditor;
```

Using the packages via npm

All the packages are also available on [npm](#) if you want to bundle them in your code.

Using the same `PlainText` example, you would install the block editor module with npm:

```
npm install @wordpress/block-editor --save
```

Once installed, you can access the component in your code using:

```
import { PlainText } from '@wordpress/block-editor';
```

[Testing JavaScript code from a specific major WordPress version](#)

There is a way to quickly install a version of the individual WordPress package used with a given WordPress major version using [npm distribution tags](#) (example for WordPress 5.8.x):

```
npm install @wordpress/block-editor@wp-5.8
```

It's also possible to update all existing WordPress packages in the project with a single command:

```
npx @wordpress/scripts packages-update --dist-tag=wp-5.8
```

All major WordPress versions starting from 5.7.x are supported (e.g., wp-5.7 or wp-6.0). Each individual dist-tag always points to the latest bug fix release for that major version line.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Package Reference](#)”

[Previous ZStack](#) [Previous: ZStack](#)

[Next @wordpress/admin-manifest](#) [Next: @wordpress/admin-manifest](#)

@wordpress/admin-manifest

[↑ Back to top](#)

Dynamically creates a Web App [manifest](#) and registers the service worker for the admin.

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](#).

First published

July 5, 2021

Last updated

April 21, 2022

Edit article

[Improve it on GitHub: @wordpress/admin-manifest”](#)

[Previous Package Reference](#) [Previous: Package Reference](#)

[Next @wordpress/a11y](#) [Next: @wordpress/a11y](#)

@wordpress/a11y

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [setup](#)
 - [speak](#)
 - [Background](#)
- [Browser support](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Accessibility utilities for WordPress.

Installation

Install the module

```
npm install @wordpress/a11y --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

setup

Create the live regions.

speak

Allows you to easily announce dynamic interface updates to screen readers using ARIA live regions. This module is inspired by the `speak` function in `wp-a11y.js`.

Usage

```
import { speak } from '@wordpress/a11y';

// For polite messages that shouldn't interrupt what screen readers are currently reading
speak( 'The message you want to send to the ARIA live region' );

// For assertive messages that should interrupt what screen readers are currently reading
speak( 'The message you want to send to the ARIA live region', 'assertive' );
```

Parameters

- `message` `string`: The message to be announced by assistive technologies.
- `ariaLive` `[string]`: The politeness level for aria-live; default: ‘polite’.

Background

For context I’ll quote [this article on WordPress.org](#) by [@joedolson](#):

Why.

In modern web development, updating discrete regions of a screen with JavaScript is common. The use of AJAX responses in modern JavaScript-based User Interfaces allows web developers to create beautiful interfaces similar to Desktop applications that don’t require pages to reload or refresh.

JavaScript can create great usability improvements for most users – but when content is updated dynamically, it has the potential to introduce accessibility issues. This is one of the steps you can take to handle that problem.

What.

When a portion of a page is updated with JavaScript, the update is usually highlighted with animation and bright colors, and is easy to see. But if you don’t have the ability to see the screen, you don’t know this has happened, unless the updated region is marked as an ARIA-live region.

If this isn’t marked, there’s no notification for screen readers. But it’s also possible that all the region content will be announced after an update, if the ARIA live region is too large. You want to provide users with just a simple, concise message.

How.

That's what `wp.a11y.speak()` is meant for. It's a simple tool that creates and appends an ARIA live notifications area to the element where developers can dispatch text messages. Assistive technologies will automatically announce any text change in this area. This ARIA live region has an ARIA role of "status" so it has an implicit aria-live value of polite and an implicit aria-atomic value of true.

This means assistive technologies will notify users of updates but generally do not interrupt the current task, and updates take low priority. If you're creating an application with higher priority updates (such as a notification that their current session is about to expire, for example), then you'll want to create your own method with an aria-live value of assertive.

[Browser support](#)

See <https://make.wordpress.org/design/handbook/design-guide/browser-support/>

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/a11y](#)

[Previous](#) [@wordpress/admin-manifest](#) [Previous: @wordpress/admin-manifest](#)
[Next](#) [@wordpress/annotations](#) [Next: @wordpress/annotations](#)

@wordpress/annotations

In this article

Table of Contents

- [Installation](#)
- [Getting Started](#)

- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Annotate content in the Gutenberg editor.

[Installation](#)

Install the module

```
npm install @wordpress/annotations --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.*

[Getting Started](#)

You need to include `wp-annotations` as a dependency of the JavaScript file in which you wish to use the Annotations API.

[Usage](#)

[See this page for more detailed usage instructions.](#)

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/annotations”](#)

[Previous: @wordpress/a11y](#) [Previous: @wordpress/a11y](#)

[Next: @wordpress/api-fetch](#) [Next: @wordpress/api-fetch](#)

@wordpress/api-fetch

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
 - [GET](#)
 - [GET with Query Args](#)
 - [POST](#)
 - [Options](#)
 - [Aborting a request](#)
 - [Middlewares](#)
 - [Built-in middlewares](#)
 - [Custom fetch handler](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Utility to make WordPress REST API requests. It's a wrapper around `window.fetch`.

[Installation](#)

Install the module

```
npm install @wordpress/api-fetch --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.*

[Usage](#)

[GET](#)

```
import apiFetch from '@wordpress/api-fetch';

apiFetch( { path: '/wp/v2/posts' } ).then( ( posts ) => {
    console.log( posts );
} );
```

[GET with Query Args](#)

```
import apiFetch from '@wordpress/api-fetch';
import { addQueryArgs } from '@wordpress/url';

const queryParams = { include: [1,2,3] }; // Return posts with ID = 1,2,3.
```

```
apiFetch( { path: addQueryArgs( '/wp/v2/posts', queryParams ) } ).then( ( posts ) => {
  console.log( posts );
} );
```

POST

```
apiFetch( {
  path: '/wp/v2/posts/1',
  method: 'POST',
  data: { title: 'New Post Title' },
} ).then( ( res ) => {
  console.log( res );
} );
```

Options

apiFetch supports and passes through all [options of the fetch global](#).

Additionally, the following options are available:

path (string)

Shorthand to be used in place of `url`, appended to the REST API root URL for the current site.

url (string)

Absolute URL to the endpoint from which to fetch.

parse (boolean, default true)

Unlike `fetch`, the Promise return value of `apiFetch` will resolve to the parsed JSON result. Disable this behavior by passing `parse` as `false`.

data (object)

Sent on POST or PUT requests only. Shorthand to be used in place of `body`, accepts an object value to be stringified to JSON.

Aborting a request

Aborting a request can be achieved through the use of [AbortController](#) in the same way as you would when using the native `fetch` API.

For legacy browsers that don't support `AbortController`, you can either:

- Provide your own polyfill of `AbortController` if you still want it to be abortable.
- Ignore it as shown in the example below.

Example

```
const controller =
  typeof AbortController === 'undefined' ? undefined : new AbortController();
```

```

apiFetch( { path: '/wp/v2/posts', signal: controller?.signal } ).catch(
  ( error ) => {
    // If the browser doesn't support AbortController then the code be
    // However, in most cases this should be fine as it can be consider
    if ( error.name === 'AbortError' ) {
      console.log( 'Request has been aborted' );
    }
  }
);
controller?.abort();

```

Middlewares

the `api-fetch` package supports middlewares. Middlewares are functions you can use to wrap the `apiFetch` calls to perform any pre/post process to the API requests.

Example

```

import apiFetch from '@wordpress/api-fetch';

apiFetch.use( ( options, next ) => {
  const start = Date.now();
  const result = next( options );
  result.then( () => {
    console.log( 'The request took ' + ( Date.now() - start ) + 'ms' );
  } );
  return result;
} );

```

Built-in middlewares

The `api-fetch` package provides built-in middlewares you can use to provide a nonce and a custom rootURL.

Nonce middleware

```

import apiFetch from '@wordpress/api-fetch';

const nonce = 'nonce value';
apiFetch.use( apiFetch.createNonceMiddleware( nonce ) );

```

The function returned by `createNonceMiddleware` includes a `nonce` property corresponding to the actively used nonce. You may also assign to this property if you have a fresh nonce value to use.

Root URL middleware

```

import apiFetch from '@wordpress/api-fetch';

const rootURL = 'http://my-wordpress-site/wp-json/';
apiFetch.use( apiFetch.createRootURLMiddleware( rootURL ) );

```

Custom fetch handler

The `api-fetch` package uses `window.fetch` for making the requests but you can use a custom fetch handler by using the `setFetchHandler` method. The custom fetch handler will receive the options passed to the `apiFetch` calls.

Example

The example below uses a custom fetch handler for making all the requests with [axios](#).

```
import apiFetch from '@wordpress/api-fetch';
import axios from 'axios';

apiFetch.setFetchHandler( ( options ) => {
    const { url, path, data, method } = options;

    return axios( {
        url: url || path,
        method,
        data,
    } );
} );
```

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/api-fetch](#)

[Previous: @wordpress/annotations](#) [Previous: @wordpress/annotations](#)
[Next: @wordpress/autop](#) [Next: @wordpress/autop](#)

@wordpress/autop

In this article

Table of Contents

- [Installation](#)
 - [API](#)
- [Contributing to this package](#)

[↑ Back to top](#)

JavaScript port of WordPress's automatic paragraph function `autop` and the `removeP` reverse behavior.

[Installation](#)

Install the module

```
npm install @wordpress/autop --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](#)

autop

Replaces double line-breaks with paragraph elements.

A group of regex replaces used to identify text formatted with newlines and replace double line-breaks with HTML paragraph tags. The remaining linebreaks after conversion become `
` tags, unless `br` is set to 'false'.

Usage

```
import { autop } from '@wordpress/autop';
autop( 'my text' ); // "<p>my text</p>"
```

Parameters

- `text` `string`: The text which has to be formatted.
- `br` `boolean`: Optional. If set, will convert all remaining line-breaks after paragraphing. Default true.

Returns

- `string`: Text which has been converted into paragraph tags.

removep

Replaces <p> tags with two line breaks. “Opposite” of autop().

Replaces <p> tags with two line breaks except where the <p> has attributes. Unifies whitespace. Indents , <dt> and <dd> for better readability.

Usage

```
import { removep } from '@wordpress/autop';
removep( '<p>my text</p>' ); // "my text"
```

Parameters

- *html string*: The content from the editor.

Returns

- *string*: The content with stripped paragraph tags.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/autop](#)

[Previous: @wordpress/api-fetch](#) [Previous: @wordpress/api-fetch](#)

[Next: @wordpress/babel-plugin-import-jsx-pragma](#) [Next: @wordpress/babel-plugin-import-jsx-pragma](#)

@wordpress/babel-plugin-import-jsx-pragma

In this article

Table of Contents

- [Installation](#)

- [Usage](#)
- [Options](#)
 - [scopeVariable](#)
 - [scopeVariableFrag](#)
 - [source](#)
 - [isDefault](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Babel transform plugin for automatically injecting an import to be used as the pragma for the [React JSX Transform plugin](#).

[JSX](#) is merely a syntactic sugar for a function call, typically to `React.createElement` when used with [React](#). As such, it requires that the function referenced by this transform be within the scope of the file where the JSX occurs. In a typical React project, this means React must be imported in any file where JSX exists.

Babel Plugin Import JSX Pragma automates this process by introducing the necessary import automatically wherever JSX exists, allowing you to use JSX in your code without thinking to ensure the transformed function is within scope. It respects existing import statements, as well as scope variable declarations.

[**Installation**](#)

Install the module to your project using [npm](#).

```
npm install @wordpress/babel-plugin-import-jsx-pragma
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[**Usage**](#)

Refer to the [Babel Plugins documentation](#) if you don't yet have experience working with Babel plugins.

Include `@wordpress/babel-plugin-import-jsx-pragma` (and [@babel/plugin-transform-react-jsx](#)) as plugins in your Babel configuration. If you don't include both you will receive errors when encountering JSX tokens.

```
// .babelrc.js
module.exports = {
  plugins: [
    '@wordpress/babel-plugin-import-jsx-pragma',
    '@babel/plugin-transform-react-jsx',
  ],
};
```

Note: `@wordpress/babel-plugin-import-jsx-pragma` is included in `@wordpress/babel-preset-default` (default preset for WordPress development) starting from v4.0.0. If you are using this preset, you shouldn't include this plugin in your Babel config.

Options

As the `@babel/plugin-transform-react-jsx` plugin offers options to customize the pragma to which the transform references, there are equivalent options to assign for customizing the imports generated.

For example, if you are using the `react` package, you may want to use the following configuration:

```
// .babelrc.js
module.exports = {
  plugins: [
    [
      '@wordpress/babel-plugin-import-jsx-pragma',
      {
        scopeVariable: 'createElement',
        scopeVariableFrag: 'Fragment',
        source: 'react',
        isDefault: false,
      },
    ],
    [
      '@babel/plugin-transform-react-jsx',
      {
        pragma: 'createElement',
        pragmaFrag: 'Fragment',
      },
    ],
  ],
};
```

scopeVariable

Type: String

Name of variable required to be in scope for use by the JSX pragma. For the default pragma of `React.createElement`, the `React` variable must be within scope.

scopeVariableFrag

Type: String

Name of variable required to be in scope for `<></>` `Fragment` JSX. Named `<Fragment />` elements expect `Fragment` to be in scope and will not add the import.

source

Type: String

The module from which the scope variable is to be imported when missing.

isDefault

Type: Boolean

Whether the scopeVariable is the default import of the source module. Note that this has no impact on scopeVariableFrag.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/babel-plugin-import-jsx-pragma”](#)

[Previous @wordpress/autop](#) [Previous: @wordpress/autop](#)

[Next @wordpress/babel-plugin-makepot](#) [Next: @wordpress/babel-plugin-makepot](#)

@wordpress/babel-plugin-makepot

In this article

Table of Contents

- [Installation](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Babel plugin used to scan JavaScript files for use of localization functions. It then compiles these into a [gettext POT formatted](#) file as a template for translation. By default the output file will be written to `gettext.pot` of the root project directory. This can be overridden using the "output" option of the plugin.

```
{  
  "plugins": [
```

```
[  
    "@wordpress/babel-plugin-makepot",  
    { "output": "languages/myplugin.pot" }  
]  
}  
}
```

Installation

Install the module:

```
npm install @wordpress/babel-plugin-makepot --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/babel-plugin-makepot](#)

[Previous: @wordpress/babel-plugin-import-jsx-pragma](#) [Previous: @wordpress/babel-plugin-import-jsx-pragma](#)

[Next: @wordpress/babel-preset-default](#) [Next: @wordpress/babel-preset-default](#)

@wordpress/babel-preset-default

In this article

Table of Contents

- [Installation](#)
 - [Usage](#)
 - [Polyfill](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Default [Babel](#) preset for WordPress development.

The preset includes configuration which enable language features and syntax extensions targeted for support by WordPress. This includes [ECMAScript proposals](#) which have reached [Stage 4 \(“Finished”\)](#), as well as the [JSX syntax extension](#). For more information, refer to the [JavaScript Coding Guidelines](#).

Installation

Install the module

```
npm install @wordpress/babel-preset-default --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

Usage

There are a number of methods to configure Babel. See [Babel’s Configuration documentation](#) for more information. To use this preset, simply reference `@wordpress/babel-preset-default` in the `presets` option in your Babel configuration.

For example, using `.babelrc`:

```
{  
  "presets": [ "@wordpress/babel-preset-default" ]  
}
```

Extending Configuration

This preset is an opinionated configuration. If you would like to add to or change this configuration, you can do so by expanding your Babel configuration to include plugins or presets which override those included through this preset. It may help to familiarize yourself [the implementation of the configuration](#) to see which specific plugins are enabled by default through this preset.

For example, if you’d like to use a new language feature proposal which has not reached the stability requirements of WordPress, you can add those as additional plugins in your Babel configuration:

```
{  
  "presets": [ "@wordpress/babel-preset-default" ],  
  "plugins": [ "@babel/plugin-proposal-class-properties" ]  
}
```

Polyfill

There is a complementary `build/polyfill.js` (minified version – `build/polyfill.min.js`) file available that polyfills ECMAScript features missing in the [browsers supported](#) by the WordPress project ([#31279](#)). It’s a drop-in replacement for the deprecated `@babel/polyfill` package, and it’s also based on [core-js](#) project.

This needs to be included before all your compiled Babel code. You can either prepend it to your compiled code or include it in a `<script>` before it.

TC39 Proposals

If you need to use a proposal that is not Stage 4, this polyfill will not automatically import those for you. You will have to import those from another polyfill like [core-js](#) individually.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/babel-preset-default”](#)

[Previous @wordpress/babel-plugin-makepot](#) [Previous: @wordpress/babel-plugin-makepot](#)
[Next @wordpress/base-styles](#) [Next: @wordpress/base-styles](#)

①@wordpress/base-styles

In this article

[Table of Contents](#)

- [Installation](#)
- [Use](#)
 - [SCSS utilities and variables](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Base SCSS utilities and variables for WordPress.

Installation

Install the module

```
npm install @wordpress/base-styles --save-dev
```

Use

SCSS utilities and variables

In your application's SCSS file, include styles like so:

```
@import 'node_modules/@wordpress/base-styles/colors';
@import 'node_modules/@wordpress/base-styles/variables';
@import 'node_modules/@wordpress/base-styles/mixins';
@import 'node_modules/@wordpress/base-styles/breakpoints';
@import 'node_modules/@wordpress/base-styles/animations';
@import 'node_modules/@wordpress/base-styles/z-index';
@import 'node_modules/@wordpress/base-styles/default-custom-properties';
```

If you use [Webpack](#) for your SCSS pipeline, you can use ~ to resolve to node_modules:

```
@import '~@wordpress/base-styles/colors';
```

To make that work with [sass](#) or [node-sass](#) NPM modules without Webpack, you'd have to use [includePaths option](#):

```
{
  "includePaths": [ "node_modules" ]
}
```

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/base-styles”](#)

@wordpress/blob

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [createBlobURL](#)
 - [downloadBlob](#)
 - [getBlobByURL](#)
 - [getBlobTypeByURL](#)
 - [isBlobURL](#)
 - [revokeBlobURL](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Blob utilities for WordPress.

[Installation](#)

Install the module

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

[API](#)

[createBlobURL](#)

Create a blob URL from a file.

Parameters

- `file` `File`: The file to create a blob URL for.

Returns

- `string`: The blob URL.

[downloadBlob](#)

Downloads a file, e.g., a text or readable stream, in the browser. Appropriate for downloading smaller file sizes, e.g., < 5 MB.

Example usage:

```
const fileContent = JSON.stringify(
  {
    title: 'My Post',
  },
  null,
  2
);
const filename = 'file.json';

downloadBlob( filename, fileContent, 'application/json' );
```

Parameters

- *filename* string: File name.
- *content* BlobPart: File content (BufferSource | Blob | string).
- *contentType* string: (Optional) File mime type. Default is ''.

[getBlobByUrl](#)

Retrieve a file based on a blob URL. The file must have been created by `createBlobURL` and not removed by `revokeBlobURL`, otherwise it will return `undefined`.

Parameters

- *url* string: The blob URL.

Returns

- File|undefined: The file for the blob URL.

[getBlobTypeByUrl](#)

Retrieve a blob type based on URL. The file must have been created by `createBlobURL` and not removed by `revokeBlobURL`, otherwise it will return `undefined`.

Parameters

- *url* string: The blob URL.

Returns

- string|undefined: The blob type.

[isBlobURL](#)

Check whether a url is a blob url.

Parameters

- *url* string|undefined: The URL.

Returns

- boolean: Is the url a blob url?

[revokeBlobURL](#)

Remove the resource and file cache from memory.

Parameters

- *url* `string`: The blob URL.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/blob”](#)

[Previous @wordpress/base-styles](#) [Previous: @wordpress/base-styles](#)
[Next @wordpress/block-directory](#) [Next: @wordpress/block-directory](#)

@wordpress/block-directory

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
- [Actions](#)
 - [addInstalledBlockType](#)
 - [clearErrorNotice](#)
 - [fetchDownloadableBlocks](#)
 - [installBlockType](#)
 - [receiveDownloadableBlocks](#)
 - [removeInstalledBlockType](#)
 - [setErrorNotice](#)
 - [setIsInstalling](#)
 - [uninstallBlockType](#)

- [Selectors](#)
 - [getDownloadableBlocks](#)
 - [getErrorNoticeForBlock](#)
 - [getErrorNotices](#)
 - [getInstalledBlockTypes](#)
 - [getNewBlockTypes](#)
 - [getUnusedBlockTypes](#)
 - [isInstalling](#)
 - [isRequestingDownloadableBlocks](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Package used to extend editor with block directory features to search and install blocks.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

Installation

Install the module

```
npm install @wordpress/block-directory --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.*

Usage

This package builds a standalone JS file. When loaded on a page with the block editor, it extends the block inserter to search for blocks from WordPress.org.

To do this, it uses the `__unstableInserterMenuExtension`, a slot-fill area hooked into the block types list. When the user runs a search and there are no results currently installed, it fires off a request to WordPress.org for matching blocks. These are listed for the user to install with a one-click process that [installs, activates, and injects the block into the post](#). When the post is saved, if the block was not used, it will be [silently uninstalled](#) to avoid clutter.

See also the API endpoints for searching WordPress.org: `/wp/v2/block-directory/search`, and installing & activating plugins: `/wp/v2/plugins/`.

Actions

The following set of dispatching action creators are available on the object returned by `wp.data.dispatch('core/block-directory')`:

[addInstalledBlockType](#)

Returns an action object used to add a block type to the “newly installed” tracking list.

Parameters

- *item Object*: The block item with the block id and name.

Returns

- *Object*: Action object.

[clearErrorNotice](#)

Sets the error notice to empty for specific block.

Parameters

- *blockId string*: The ID of the block plugin. eg: my-block

Returns

- *Object*: Action object.

[fetchDownloadableBlocks](#)

Returns an action object used in signalling that the downloadable blocks have been requested and are loading.

Parameters

- *filterValue string*: Search string.

Returns

- *Object*: Action object.

[installBlockType](#)

Action triggered to install a block plugin.

Parameters

- *block Object*: The block item returned by search.

Returns

- *boolean*: Whether the block was successfully installed & loaded.

[receiveDownloadableBlocks](#)

Returns an action object used in signalling that the downloadable blocks have been updated.

Parameters

- *downloadableBlocks Array*: Downloadable blocks.
- *filterValue string*: Search string.

Returns

- `Object`: Action object.

[removeInstalledBlockType](#)

Returns an action object used to remove a block type from the “newly installed” tracking list.

Parameters

- `item string`: The block item with the block id and name.

Returns

- `Object`: Action object.

[setErrorNotice](#)

Sets an error notice to be displayed to the user for a given block.

Parameters

- `blockId string`: The ID of the block plugin. eg: my-block
- `message string`: The message shown in the notice.
- `isFatal boolean`: Whether the user can recover from the error.

Returns

- `Object`: Action object.

[setIsInstalling](#)

Returns an action object used to indicate install in progress.

Parameters

- `blockId string`:
- `isInstalling boolean`:

Returns

- `Object`: Action object.

[uninstallBlockType](#)

Action triggered to uninstall a block plugin.

Parameters

- `block Object`: The blockType object.

Selectors

The following selectors are available on the object returned by `wp.data.select('core/block-directory')`:

getDownloadableBlocks

Returns the available uninstalled blocks.

Parameters

- `state Object`: Global application state.
- `filterValue string`: Search string.

Returns

- `Array`: Downloadable blocks.

getErrorNoticeForBlock

Returns the error notice for a given block.

Parameters

- `state Object`: Global application state.
- `blockId string`: The ID of the block plugin. eg: my-block

Returns

- `string | boolean`: The error text, or false if no error.

getErrorNotices

Returns all block error notices.

Parameters

- `state Object`: Global application state.

Returns

- `Object`: Object with error notices.

getInstalledBlockTypes

Returns the block types that have been installed on the server in this session.

Parameters

- `state Object`: Global application state.

Returns

- `Array`: Block type items

[getNewBlockTypes](#)

Returns block types that have been installed on the server and used in the current post.

Parameters

- *state Object*: Global application state.

Returns

- *Array*: Block type items.

[getUnusedBlockTypes](#)

Returns the block types that have been installed on the server but are not used in the current post.

Parameters

- *state Object*: Global application state.

Returns

- *Array*: Block type items.

[isInstalling](#)

Returns true if a block plugin install is in progress.

Parameters

- *state Object*: Global application state.
- *blockId string*: Id of the block.

Returns

- *boolean*: Whether this block is currently being installed.

[isRequestingDownloadableBlocks](#)

Returns true if application is requesting for downloadable blocks.

Parameters

- *state Object*: Global application state.
- *filterValue string*: Search string.

Returns

- *boolean*: Whether a request is in progress for the blocks list.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/block-directory](#)

[Previous](#) [@wordpress/blob](#) [Previous: @wordpress/blob](#)

[Next](#) [@wordpress/block-editor](#) [Next: @wordpress/block-editor](#)

@wordpress/block-editor

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [API](#)
 - [AlignmentControl](#)
 - [AlignmentToolbar](#)
 - [Autocomplete](#)
 - [BlockAlignmentControl](#)
 - [BlockAlignmentToolbar](#)
 - [BlockBreadcrumb](#)
 - [BlockCanvas](#)
 - [BlockColorsStyleSelector](#)
 - [BlockContextProvider](#)
 - [BlockControls](#)
 - [BlockEdit](#)
 - [BlockEditorKeyboardShortcuts](#)
 - [BlockEditorProvider](#)
 - [BlockFormatControls](#)
 - [BlockIcon](#)
 - [BlockInspector](#)
 - [BlockList](#)
 - [BlockMover](#)
 - [BlockNavigationDropdown](#)
 - [BlockPreview](#)
 - [BlockSelectionClearer](#)
 - [BlockSettingsMenu](#)
 - [BlockSettingsMenuControls](#)

- [BlockStyles](#)
- [BlockTitle](#)
- [BlockToolbar](#)
- [BlockTools](#)
- [BlockVerticalAlignmentControl](#)
- [BlockVerticalAlignmentToolbar](#)
- [ButtonBlockAppender](#)
- [ButtonBlockerAppender](#)
- [ColorPalette](#)
- [ColorPaletteControl](#)
- [ContrastChecker](#)
- [CopyHandler](#)
- [createCustomColorsHOC](#)
- [DefaultBlockAppender](#)
- [FontSizePicker](#)
- [getColorClassName](#)
- [getColorObjectByAttributeValues](#)
- [getColorObjectByColorValue](#)
- [getComputedFluidTypographyValue](#)
- [getCustomValueFromPreset](#)
- [getFontSize](#)
- [getFontSizeClass](#)
- [getFontSizeObjectByValue](#)
- [getGradientSlugByValue](#)
- [getGradientValueBySlug](#)
- [getPxFromCssUnit](#)
- [getSpacingPresetCssVar](#)
- [getTypographyClassesAndStyles](#)
- [HeadingLevelDropdown](#)
- [HeightControl](#)
- [InnerBlocks](#)
- [Inserter](#)
- [InspectorAdvancedControls](#)
- [InspectorControls](#)
- [isValueSpacingPreset](#)
- [JustifyContentControl](#)
- [JustifyToolbar](#)
- [LineHeightControl](#)
- [MediaPlaceholder](#)
- [MediaReplaceFlow](#)
- [MediaUpload](#)
- [MediaUploadCheck](#)
- [MultiSelectScrollIntoView](#)
- [NavigableToolbar](#)
- [ObserveTyping](#)
- [PanelColorSettings](#)
- [PlainText](#)
- [privateApis](#)
- [RecursionProvider](#)
- [ReusableBlocksRenameHint](#)
- [RichText](#)
- [RichTextShortcut](#)
- [RichTextToolbarButton](#)
- [SETTINGS_DEFAULTS](#)

- [SkipToSelectedBlock](#)
- [store](#)
- [storeConfig](#)
- [ToolSelector](#)
- [transformStyles](#)
- [Typewriter](#)
- [URLInput](#)
- [URLInputButton](#)
- [URLPopover](#)
- [useBlockCommands](#)
- [useBlockDisplayInformation](#)
- [useBlockEditContext](#)
- [useBlockEditingStyle](#)
- [useBlockProps](#)
- [useCachedTruthy](#)
- [useHasRecursion](#)
- [useInnerBlocksProps](#)
- [useSetting](#)
- [useSettings](#)
- [Warning](#)
- [withColorContext](#)
- [withColors](#)
- [withFontSizes](#)
- [WritingFlow](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This module allows you to create and use standalone block editors.

Installation

Install the module

```
npm install @wordpress/block-editor --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.*

Usage

```
import { useState } from 'react';
import {
    BlockEditorProvider,
    BlockList,
    WritingFlow,
} from '@wordpress/block-editor';

function MyEditorComponent() {
    const [ blocks, updateBlocks ] = useState( [] );
    // ...
}
```

```

        return (
          <BlockEditorProvider
            value={ blocks }
            onInput={ ( blocks ) => updateBlocks( blocks ) }
            onChange={ ( blocks ) => updateBlocks( blocks ) }
          >
            <BlockCanvas height="400px" />
          </BlockEditorProvider>
        );
      }

// Make sure to load the block editor stylesheets too
// import '@wordpress/components/build-style/style.css';
// import '@wordpress/block-editor/build-style/style.css';

```

In this example, we're instantiating a block editor. A block editor is composed by a `BlockEditorProvider` wrapper component where you pass the current array of blocks and on each change the `onInput` or `onChange` callbacks are called depending on whether the change is considered persistent or not.

Inside `BlockEditorProvider`, you can nest any of the available `@wordpress/block-editor` UI components to build the UI of your editor.

In the example above we're rendering the `BlockList` to show and edit the block list. For instance we could add a custom sidebar and use the `BlockInspector` component to be able to edit the advanced settings for the currently selected block. (See the [API](#) for the list of all the available components).

The `BlockTools` component is used to render the toolbar for a selected block.

In the example above, there's no registered block type, in order to use the block editor successfully make sure to register some block types. For instance, registering the core block types can be done like so:

```

import { registerCoreBlocks } from '@wordpress/block-library';

registerCoreBlocks();

// Make sure to load the block stylesheets too
// import '@wordpress/block-library/build-style/style.css';
// import '@wordpress/block-library/build-style/editor.css';
// import '@wordpress/block-library/build-style/theme.css';

```

[API](#)

Any components in this package that have a counterpart in [@wordpress/components](#) are an extension of those components.

Unless you're [creating an editor](#), it is recommended that the components in `@wordpress/components` should be used rather than the ones in this package as these components have been customized for use in an editor and may result in unexpected behaviour if used outside of this context.

[AlignmentControl](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/alignment-control/README.md>

[AlignmentToolbar](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/alignment-control/README.md>

[Autocomplete](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/autocomplete/README.md>

[BlockAlignmentControl](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-alignment-control/README.md>

[BlockAlignmentToolbar](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-alignment-control/README.md>

[BlockBreadcrumb](#)

Block breadcrumb component, displaying the hierarchy of the current block selection as a breadcrumb.

Parameters

- `props Object`: Component props.
- `props.rootLabelText string`: Translated label for the root element of the breadcrumb trail.

Returns

- `Element`: Block Breadcrumb.

[BlockCanvas](#)

BlockCanvas component is a component used to display the canvas of the block editor. What we call the canvas is an iframe containing the block list that you can manipulate. The component is

also responsible of wiring up all the necessary hooks to enable the keyboard navigation across blocks in the editor and inject content styles into the iframe.

Usage

```
function MyBlockEditor() {
  const [ blocks, updateBlocks ] = useState( [] );
  return (
    <BlockEditorProvider
      value={ blocks }
      onInput={ updateBlocks }
      onChange={ persistBlocks }
    >
    <BlockCanvas height="400px" />
  </BlockEditorProvider>
);
}
```

Parameters

- `props Object`: Component props.
- `props.height string`: Canvas height, defaults to 300px.
- `props.styles Array`: Content styles to inject into the iframe.
- `props.children Element`: Content of the canvas, defaults to the BlockList component.

Returns

- `Element`: Block Breadcrumb.

[BlockColorsStyleSelector](#)

Undocumented declaration.

[BlockContextProvider](#)

Component which merges passed value with current consumed block context.

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-context/README.md>

Parameters

- `props BlockContextProviderProps`:

[BlockControls](#)

Undocumented declaration.

[BlockEdit](#)

Undocumented declaration.

BlockEditorKeyboardShortcuts

Undocumented declaration.

BlockEditorProvider

Undocumented declaration.

BlockFormatControls

Undocumented declaration.

BlockIcon

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-icon/README.md>

BlockInspector

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-inspector/README.md>

BlockList

Undocumented declaration.

BlockMover

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-mover/README.md>

BlockNavigationDropdown

Undocumented declaration.

BlockPreview

BlockPreview renders a preview of a block or array of blocks.

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-preview/README.md>

Parameters

- *preview Object*: options for how the preview should be shown

- *preview.blocks* `Array | Object`: A block instance (object) or an array of blocks to be previewed.
- *preview.viewportWidth* `number`: Width of the preview container in pixels. Controls at what size the blocks will be rendered inside the preview. Default: 700.

Returns

- `Component`: The component to be rendered.

[BlockSelectionClearer](#)

Undocumented declaration.

[BlockSettingsMenu](#)

Undocumented declaration.

[BlockSettingsMenuControls](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-settings-menu-controls/README.md>

Parameters

- *props* `Object`: Fill props.

Returns

- `Element`: Element.

[BlockStyles](#)

Undocumented declaration.

[BlockTitle](#)

Renders the block's configured title as a string, or empty if the title cannot be determined.

Usage

```
<BlockTitle
  clientId="afd1cb17-2c08-4e7a-91be-007ba7ddc3a1"
  maxLength={ 17 }
/>
```

Parameters

- *props* `Object`:
- *props.clientId* `string`: Client ID of block.
- *props.maxLength* `number | undefined`: The maximum length that the block title string may be before truncated.
- *props.context* `string | undefined`: The context to pass to `getBlockLabel`.

Returns

- `JSX.Element`: Block title.

[BlockToolbar](#)

Renders the block toolbar.

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-toolbar/README.md>

Parameters

- `props Object`: Components props.
- `props.hideDragHandle boolean`: Show or hide the Drag Handle for drag and drop functionality.
- `props.variant string`: Style variant of the toolbar, also passed to the Dropdowns rendered from Block Toolbar Buttons.

[BlockTools](#)

Renders block tools (the block toolbar, select/navigation mode toolbar, the insertion point and a slot for the inline rich text toolbar). Must be wrapped around the block content and editor styles wrapper or iframe.

Parameters

- `$0 Object`: Props.
- `$0.children Object`: The block content and style container.
- `$0.__unstableContentRef Object`: Ref holding the content scroll container.

[BlockVerticalAlignmentControl](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-vertical-alignment-control/README.md>

[BlockVerticalAlignmentToolbar](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/block-vertical-alignment-control/README.md>

[ButtonBlockAppender](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/button-block-appender/README.md>

[ButtonBlockerAppender](#)

Deprecated

Use `ButtonBlockAppender` instead.

[ColorPalette](#)

Undocumented declaration.

[ColorPaletteControl](#)

Undocumented declaration.

[ContrastChecker](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/contrast-checker/README.md>

[CopyHandler](#)

Deprecated

Parameters

- `props Object`:

[createCustomColorsHOC](#)

A higher-order component factory for creating a ‘withCustomColors’ HOC, which handles color logic for class generation color value, retrieval and color attribute setting.

Use this higher-order component to work with a custom set of colors.

Usage

```
const CUSTOM_COLORS = [
  { name: 'Red', slug: 'red', color: '#ff0000' },
  { name: 'Blue', slug: 'blue', color: '#0000ff' },
];
const withCustomColors = createCustomColorsHOC( CUSTOM_COLORS );
// ...
export default compose(
  withCustomColors( 'backgroundColor', 'borderColor' ),
  MyColorfulComponent
);
```

Parameters

- `colorsArray Array`: The array of color objects (name, slug, color, etc...).

Returns

- Function: Higher-order component.

[DefaultBlockAppender](#)

Undocumented declaration.

[FontSizePicker](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/font-sizes/README.md>

[getColorClassName](#)

Returns a class based on the context a color is being used and its slug.

Parameters

- *colorContextName* string: Context/place where color is being used e.g: background, text etc...
- *colorSlug* string: Slug of the color.

Returns

- ?string: String with the class corresponding to the color in the provided context.
Returns undefined if either colorContextName or colorSlug are not provided.

[getColorObjectByAttributeValues](#)

Provided an array of color objects as set by the theme or by the editor defaults, and the values of the defined color or custom color returns a color object describing the color.

Parameters

- *colors* Array: Array of color objects as set by the theme or by the editor defaults.
- *definedColor* ?string: A string containing the color slug.
- *customColor* ?string: A string containing the customColor value.

Returns

- ?Object: If definedColor is passed and the name is found in colors, the color object exactly as set by the theme or editor defaults is returned. Otherwise, an object that just sets the color is defined.

[getColorObjectByColorValue](#)

Provided an array of color objects as set by the theme or by the editor defaults, and a color value returns the color object matching that value or undefined.

Parameters

- *colors* `Array`: Array of color objects as set by the theme or by the editor defaults.
- *colorValue* `?string`: A string containing the color value.

Returns

- `?Object`: Color object included in the colors array whose color property equals *colorValue*. Returns undefined if no color object matches this requirement.

[getComputedFluidTypographyValue](#)

Computes a fluid font-size value that uses clamp(). A minimum and maximum font size OR a single font size can be specified.

If a single font size is specified, it is scaled up and down using a logarithmic scale.

Usage

```
// Calculate fluid font-size value from a minimum and maximum value.  
const fontSize = getComputedFluidTypographyValue( {  
    minimumFontSize: '20px',  
    maximumFontSize: '45px',  
} );  
// Calculate fluid font-size value from a single font size.  
const fontSize = getComputedFluidTypographyValue( {  
    fontSize: '30px',  
} );
```

Parameters

- *args* `Object`:
- *args.minimumViewportWidth* `?string`: Minimum viewport size from which type will have fluidity. Optional if *fontSize* is specified.
- *args.maximumViewportWidth* `?string`: Maximum size up to which type will have fluidity. Optional if *fontSize* is specified.
- *args.fontSize* `[string | number]`: Size to derive *maximumFontSize* and *minimumFontSize* from, if necessary. Optional if *minimumFontSize* and *maximumFontSize* are specified.
- *args.maximumFontSize* `?string`: Maximum font size for any clamp() calculation. Optional.
- *args.minimumFontSize* `?string`: Minimum font size for any clamp() calculation. Optional.
- *args.scaleFactor* `?number`: A scale factor to determine how fast a font scales within boundaries. Optional.
- *args.minimumFontSizeLimit* `?string`: The smallest a calculated font size may be. Optional.

Returns

- `string | null`: A font-size value using clamp().

[getCustomValueFromPreset](#)

Converts a spacing preset into a custom value.

Parameters

- *value* `string`: Value to convert
- *spacingSizes* `Array`: Array of the current spacing preset objects

Returns

- `string`: Mapping of the spacing preset to its equivalent custom value.

[getFontSize](#)

Returns the font size object based on an array of named font sizes and the `namedFontSize` and `customFontSize` values. If `namedFontSize` is undefined or not found in `fontSizes` an object with just the `size` value based on `customFontSize` is returned.

Parameters

- *fontSizes* `Array`: Array of font size objects containing at least the “name” and “size” values as properties.
- *fontSizeAttribute* `?string`: Content of the font size attribute (slug).
- *customFontSizeAttribute* `?number`: Contents of the custom font size attribute (value).

Returns

- `?Object`: If `fontSizeAttribute` is set and an equal slug is found in `fontSizes` it returns the font size object for that slug. Otherwise, an object with just the `size` value based on `customFontSize` is returned.

[getFontSizeClass](#)

Returns a class based on `fontSizeName`.

Parameters

- *fontSizeSlug* `string`: Slug of the `fontSize`.

Returns

- `string | undefined`: String with the class corresponding to the `fontSize` passed. The class is generated by appending ‘has-‘ followed by `fontSizeSlug` in kebabCase and ending with ‘-font-size’.

[getFontSizeObjectByValue](#)

Returns the corresponding font size object for a given value.

Parameters

- *fontSizes* `Array`: Array of font size objects.
- *value* `number`: Font size value.

Returns

- `Object`: Font size object.

[getGradientSlugByValue](#)

Retrieves the gradient slug per slug.

Parameters

- `gradients Array`: Gradient Palette
- `value string`: Gradient value

Returns

- `string`: Gradient slug.

[getGradientValueBySlug](#)

Retrieves the gradient value per slug.

Parameters

- `gradients Array`: Gradient Palette
- `slug string`: Gradient slug

Returns

- `string`: Gradient value.

[getPxFromCssUnit](#)

Deprecated

This function was accidentally exposed for mobile/native usage.

Returns

- `string`: Empty string.

[getSpacingPresetCssVar](#)

Converts a spacing preset into a custom value.

Parameters

- `value string`: Value to convert.

Returns

- `string | undefined`: CSS var string for given spacing preset value.

[getTypographyClassesAndStyles](#)

Provides the CSS class names and inline styles for a block's typography support attributes.

Parameters

- *attributes Object*: Block attributes.
- *settings Object | boolean*: Merged theme.json settings

Returns

- *Object*: Typography block support derived CSS classes & styles.

[HeadingLevelDropdown](#)

Dropdown for selecting a heading level (1 through 6) or paragraph (0).

Parameters

- *props WPHeadingLevelDropdownProps*: Component props.

Returns

- *ComponentType*: The toolbar.

[HeightControl](#)

HeightControl renders a linked unit control and range control for adjusting the height of a block.

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/height-control/README.md>

Parameters

- *props Object*:
- *props.label ? string*: A label for the control.
- *props.onChange (value: string) => void*: Called when the height changes.
- *props.value string*: The current height value.

Returns

- *Component*: The component to be rendered.

[InnerBlocks](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/inner-blocks/README.md>

Inserter

Undocumented declaration.

InspectorAdvancedControls

Undocumented declaration.

InspectorControls

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/inspector-controls/README.md>

isValueSpacingPreset

Checks is given value is a spacing preset.

Parameters

- `value string`: Value to check

Returns

- `boolean`: Return true if value is string in format var:preset|spacing|.

JustifyContentControl

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/justify-content-control/README.md>

JustifyToolbar

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/justify-content-control/README.md>

LineHeightControl

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/line-height-control/README.md>

MediaPlaceholder

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/media-placeholder/README.md>

MediaReplaceFlow

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/media-replace-flow/README.md>

MediaUpload

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/media-upload/README.md>

MediaUploadCheck

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/media-upload/README.md>

MultiSelectScrollIntoView

Deprecated

Scrolls the multi block selection end into view if not in view already. This is important to do after selection by keyboard.

NavigableToolbar

Undocumented declaration.

ObserveTyping

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/observe-typing/README.md>

PanelColorSettings

Undocumented declaration.

PlainText

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/plain-text/README.md>

privateApis

Private @wordpress/block-editor APIs.

RecursionProvider

A React context provider for use with the `useHasRecursion` hook to prevent recursive renders.

Wrap block content with this provider and provide the same `uniqueId` prop as used with `useHasRecursion`.

Parameters

- `props Object`:
- `props.uniqueId *`: Any value that acts as a unique identifier for a block instance.
- `props.blockName string`: Optional block name.
- `props.children JSX.Element`: React children.

Returns

- `JSX.Element`: A React element.

ReusableBlocksRenameHint

Undocumented declaration.

RichText

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/rich-text/README.md>

RichTextShortcut

Undocumented declaration.

RichTextToolbarButton

Undocumented declaration.

SETTINGS_DEFAULTS

The default editor settings

Type Definition

- `SETTINGS_DEFAULT Object`

Properties

- `alignWide boolean`: Enable/Disable Wide/Full Alignments
- `supportsLayout boolean`: Enable/disable layouts support in container blocks.
- `imageEditing boolean`: Image Editing settings set to false to disable.
- `imageSizes Array`: Available image sizes
- `maxWidth number`: Max width to constraint resizing
- `allowedBlockTypes boolean|Array`: Allowed block types
- `hasFixedToolbar boolean`: Whether or not the editor toolbar is fixed
- `distractionFree boolean`: Whether or not the editor UI is distraction free
- `focusMode boolean`: Whether the focus mode is enabled or not
- `styles Array`: Editor Styles
- `keepCaretInsideBlock boolean`: Whether caret should move between blocks in edit mode
- `bodyPlaceholder string`: Empty post placeholder
- `titlePlaceholder string`: Empty title placeholder
- `canLockBlocks boolean`: Whether the user can manage Block Lock state
- `codeEditingEnabled boolean`: Whether or not the user can switch to the code editor
- `generateAnchors boolean`: Enable/Disable auto anchor generation for Heading blocks
- `enableOpenverseMediaCategory boolean`: Enable/Disable the Openverse media category in the inserter.
- `clearBlockSelection boolean`: Whether the block editor should clear selection on mousedown when a block is not clicked.
- `__experimentalCanUserUseUnfilteredHTML boolean`: Whether the user should be able to use unfiltered HTML or the HTML should be filtered e.g., to remove elements considered insecure like iframes.
- `__experimentalBlockDirectory boolean`: Whether the user has enabled the Block Directory
- `__experimentalBlockPatterns Array`: Array of objects representing the block patterns
- `__experimentalBlockPatternCategories Array`: Array of objects representing the block pattern categories
- `__unstableGalleryWithImageBlocks boolean`: Whether the user has enabled the refactored gallery block which uses InnerBlocks

SkipToSelectedBlock

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/skip-to-selected-block/README.md>

store

Store definition for the block editor namespace.

Related

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

[storeConfig](#)

Block editor data store configuration.

Related

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

[ToolSelector](#)

Undocumented declaration.

[transformStyles](#)

Applies a series of CSS rule transforms to wrap selectors inside a given class and/or rewrite URLs depending on the parameters passed.

Parameters

- `styles EditorStyle[]`: CSS rules.
- `wrapperSelector string`: Wrapper selector.

Returns

- `Array`: converted rules.

Type Definition

- `EditorStyle Object`

Properties

- `css string`: the CSS block(s), as a single string.
- `baseURL ?string`: the base URL to be used as the reference when rewriting urls.
- `ignoredSelectors ?string []`: the selectors not to wrap.

[Typewriter](#)

Ensures that the text selection keeps the same vertical distance from the viewport during keyboard events within this component. The vertical distance can vary. It is the last clicked or scrolled to position.

[URLInput](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/url-input/README.md>

[URLInputButton](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/url-input/README.md>

[URLPopover](#)

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/url-popover/README.md>

[useBlockCommands](#)

Undocumented declaration.

[useBlockDisplayInformation](#)

Hook used to try to find a matching block variation and return the appropriate information for display reasons. In order to try to find a match we need to things: 1. Block's client id to extract it's current attributes. 2. A block variation should have set `isActive` prop to a proper function.

If for any reason a block variation match cannot be found, the returned information come from the Block Type. If no blockType is found with the provided clientId, returns null.

Parameters

- `clientId` `string`: Block's client id.

Returns

- `?WPBlockDisplayInformation`: Block's display information, or `null` when the block or its type not found.

[useBlockEditContext](#)

The `useBlockEditContext` hook provides information about the block this hook is being used in. It returns an object with the `name`, `isSelected` state, and the `clientId` of the block. It is useful if you want to create custom hooks that need access to the current blocks `clientId` but don't want to rely on the data getting passed in as a parameter.

Returns

- `Object`: Block edit context

[useBlockEditingStyle](#)

Allows a block to restrict the user interface that is displayed for editing that block and its inner blocks.

Usage

```
function MyBlock( { attributes, setAttributes } ) {
  useBlockEditMode( 'disabled' );
  return <div { ...useBlockProps() }></div>;
}
```

mode can be one of three options:

- 'disabled': Prevents editing the block entirely, i.e. it cannot be selected.
- 'contentOnly': Hides all non-content UI, e.g. auxiliary controls in the toolbar, the block movers, block settings.
- 'default': Allows editing the block as normal.

The mode is inherited by all of the block's inner blocks, unless they have their own mode.

If called outside of a block context, the mode is applied to all blocks.

Parameters

- *mode* ?BlockEditMode: The editing mode to apply. If undefined, the current editing mode is not changed.

Returns

- BlockEditMode: The current editing mode.

useBlockProps

This hook is used to lightly mark an element as a block element. The element should be the outermost element of a block. Call this hook and pass the returned props to the element to mark as a block. If you define a ref for the element, it is important to pass the ref to this hook, which the hook in turn will pass to the component through the props it returns. Optionally, you can also pass any other props through this hook, and they will be merged and returned.

Use of this hook on the outermost element of a block is required if using API >= v2.

Usage

```
import { useBlockProps } from '@wordpress/block-editor';

export default function Edit() {

  const blockProps = useBlockProps(
    className: 'my-custom-class',
    style: {
      color: '#222222',
      backgroundColor: '#eeeeee'
    }
  )

  return (
    <div { ...blockProps }>
      </div>
  )
}
```

```
    )  
}
```

Parameters

- *props Object*: Optional. Props to pass to the element. Must contain the ref if one is defined.
- *options Object*: Options for internal use only.
- *options.__unstableIsHtml boolean*:

Returns

- *Object*: Props to pass to the element to mark as a block.

[useCachedTruthy](#)

Keeps an up-to-date copy of the passed value and returns it. If value becomes falsy, it will return the last truthy copy.

Parameters

- *value any*:

Returns

- *any*: value

[useHasRecursion](#)

A React hook for keeping track of blocks previously rendered up in the block tree. Blocks susceptible to recursion can use this hook in their `Edit` function to prevent said recursion.

Use this with the `RecursionProvider` component, using the same `uniqueId` value for both the hook and the provider.

Parameters

- *uniqueId **: Any value that acts as a unique identifier for a block instance.
- *blockName string*: Optional block name.

Returns

- *boolean*: A boolean describing whether the provided id has already been rendered.

[useInnerBlocksProps](#)

This hook is used to lightly mark an element as an inner blocks wrapper element. Call this hook and pass the returned props to the element to mark as an inner blocks wrapper, automatically rendering inner blocks as children. If you define a ref for the element, it is important to pass the ref to this hook, which the hook in turn will pass to the component through the props it returns. Optionally, you can also pass any other props through this hook, and they will be merged and returned.

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/inner-blocks/README.md>

Parameters

- *props Object*: Optional. Props to pass to the element. Must contain the ref if one is defined.
- *options Object*: Optional. Inner blocks options.

[useSetting](#)

Deprecated 6.5.0 Use `useSettings` instead.

Hook that retrieves the given setting for the block instance in use.

It looks up the setting first in the block instance hierarchy. If none is found, it'll look it up in the block editor settings.

Usage

```
const isEnabled = useSetting( 'typography.dropCap' );
```

Parameters

- *path string*: The path to the setting.

Returns

- *any*: Returns the value defined for the setting.

[useSettings](#)

Hook that retrieves the given settings for the block instance in use.

It looks up the settings first in the block instance hierarchy. If none are found, it'll look them up in the block editor settings.

Usage

```
const [ fixed, sticky ] = useSettings( 'position.fixed', 'position.sticky' );
```

Parameters

- *paths string []*: The paths to the settings.

Returns

- *any []*: Returns the values defined for the settings.

Warning

Related

- <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/warning/README.md>

withColorContext

Undocumented declaration.

withColors

A higher-order component, which handles color logic for class generation color value, retrieval and color attribute setting.

For use with the default editor/theme color palette.

Usage

```
export default compose(
  withColors( 'backgroundColor', { textColor: 'color' } ),
  MyColorfulComponent
);
```

Parameters

- *colorTypes . . . (Object|string)*: The arguments can be strings or objects. If the argument is an object, it should contain the color attribute name as key and the color context as value. If the argument is a string the value should be the color attribute name, the color context is computed by applying a kebab case transform to the value. Color context represents the context/place where the color is going to be used. The class name of the color is generated using ‘has’ followed by the color name and ending with the color context all in kebab case e.g: has-green-background-color.

Returns

- **Function**: Higher-order component.

withFontSizes

Higher-order component, which handles font size logic for class generation, font size value retrieval, and font size change handling.

Parameters

- *fontSizeNames . . . (Object|string)*: The arguments should all be strings. Each string contains the font size attribute name e.g: ‘fontSize’.

Returns

- **Function**: Higher-order component.

WritingFlow

Handles selection and navigation across blocks. This component should be wrapped around BlockList.

Parameters

- *props Object*: Component properties.
- *props.children Element*: Children to be rendered.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/block-editor”](#)

[Previous @wordpress/block-directory](#) Previous: [@wordpress/block-directory](#)

Next [@wordpress/block-library](#) Next: [@wordpress/block-library](#)

@wordpress/block-library

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [registerCoreBlocks](#)
- [Registering individual blocks](#)
- [Contributing to this package](#)
 - [Adding new blocks](#)
 - [Naming convention for PHP functions](#)

[↑ Back to top](#)

Block library for the WordPress editor.

Installation

Install the module

```
npm install @wordpress/block-library --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

registerCoreBlocks

Function to register core blocks provided by the block editor.

Usage

```
import { registerCoreBlocks } from '@wordpress/block-library';

registerCoreBlocks();
```

Parameters

- *blocks* Array: An optional array of the core blocks being registered.

Registering individual blocks

1. When you only care about registering the block when file gets imported:

```
import '@wordpress/block-library/build-module/verse/init';
```

2. When you want to use the reference to the block after it gets automatically registered:

```
import verseBlock from '@wordpress/block-library/build-module/verse/init';
```

3. When you need a full control over when the block gets registered:

```
import { init } from '@wordpress/block-library/build-module/verse';
const verseBlock = init();
```

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](#).

[Adding new blocks](#)

⚠ Adding new blocks to this package **requires** additional steps!

1. Do not forget to register a new core block in the [index.js](#) file of this package. For example, if you were to add the new core block called `core/blinking-paragraph`, you would have to add something like:

```
// packages/block-library/src/index.js
import * as blinkingParagraph from './blinking-paragraph';
```

Then add `blinkingParagraph` to the list in the `getAllBlocks()` function.

If it's experimental, add the following property to `block.json`:

```
{
  "__experimental": "true"
}
```

2. Register the block in the `gutenberg_reregister_core_block_types()` function of the [lib/blocks.php](#) file. Add it to the `block_folders` array if it's a [static block](#) or to the `block_names` array if it's a [dynamic block](#).

3. Add `init.js` file to the directory of the new block:

```
/**
 * Internal dependencies
 */
import { init } from './';

export default init();
```

This file is used when using the option to register individual block from the `@wordpress/block-library` package.

4. If a `view.js` file (or a file prefixed with `view`, e.g. `view-example.js`) is present in the block's directory, this file will be built along other assets, making it available to load from the browser. You only need to reference a `view.min.js` (notice the different file extension) file in the `block.json` file as follows:

```
{
  "viewScript": "file:./view.min.js"
}
```

This file will get automatically loaded when the static block is present on the front end. For dynamic block, you need to manually enqueue the view script in `render_callback` of the block, example:

```
function render_block_core_blinking_paragraph( $attributes, $content )
{
    $should_load_view_script = ! empty( $attributes['isInteractive'] )
    if ( $should_load_view_script ) {
        wp_enqueue_script( 'wp-block-blinking-paragraph-view' );
    }
}
```

```
    }

    return $content;
}
```

Naming convention for PHP functions

All PHP function names declared within the subdirectories of the `packages/block-library/src/` directory should start with one of the following prefixes:

- `block_core_<directory_name>`
- `render_block_core_<directory_name>`
- `register_block_core_<directory_name>`

In this context, `<directory_name>` represents the name of the directory where the corresponding `.php` file is located.

The directory name is converted to lowercase, and any characters except for letters and digits are replaced with underscores.

Example:

For the PHP functions declared in the `packages/block-library/src/my-block/index.php` file, the correct prefixes would be:

- `block_core_my_block`
- `render_block_core_my_block`
- `register_block_core_my_block`

Using plugin-specific prefixes/suffixes

Unlike in [PHP code in the /lib directory](#), you should generally avoid applying plugin-specific prefixes/suffixes such as `gutenberg_` to functions and other code in block PHP files.

There are times, however, when blocks may need to use Gutenberg functions even when a Core-equivalent exists, for example, where a Gutenberg function relies on code that is only available in the plugin.

In such cases, you can use the corresponding Core `wp_` function in the block PHP code, and add its name to [a list of prefixed functions in the Webpack configuration file](#).

At build time, Webpack will search for `wp_` functions in that list and replace them with their `gutenberg_` equivalents. This process ensures that the plugin calls the `gutenberg_` functions, but the block will still call the Core `wp_` function when updates are back ported.

Webpack assumes that, prefixes aside, the functions' names are identical:
`wp_get_something_useful()` will be replaced with
`gutenberg_get_something_useful()`.

First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: @wordpress/block-library”](#)

[Previous @wordpress/block-editor](#) [Previous: @wordpress/block-editor](#)

[Next @wordpress/block-serialization-default-parser](#) [Next: @wordpress/block-serialization-default-parser](#)

@wordpress/block-serialization-default-parser

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [parse](#)
- [Theory](#)
 - [What is different about this one from the spec-parser?](#)
 - [How does it work?](#)
 - [I meant, how does it perform?](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This library contains the default block serialization parser implementations for WordPress documents. It provides native PHP and JavaScript parsers that implement the specification from [@wordpress/block-serialization-spec-parser](#) and which normally operates on the document stored in `post_content`.

[Installation](#)

Install the module

```
npm install @wordpress/block-serialization-default-parser --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](#)

[parse](#)

Parser function, that converts input HTML into a block based structure.

Usage

Input post:

```
<!-- wp:columns {"columns":3} -->
<div class="wp-block-columns has-3-columns">
    <!-- wp:column -->
    <div class="wp-block-column">
        <!-- wp:paragraph -->
        <p>Left</p>
        <!-- /wp:paragraph -->
    </div>
    <!-- /wp:column -->

    <!-- wp:column -->
    <div class="wp-block-column">
        <!-- wp:paragraph -->
        <p><strong>Middle</strong></p>
        <!-- /wp:paragraph -->
    </div>
    <!-- /wp:column -->

    <!-- wp:column -->
    <div class="wp-block-column"></div>
    <!-- /wp:column -->
</div>
<!-- /wp:columns -->
```

Parsing code:

```
import { parse } from '@wordpress/block-serialization-default-parser';

parse( post ) ===
[
    {
        blockName: 'core/columns',
        attrs: {
            columns: 3,
        },
        innerBlocks: [
            {
                blockName: 'core/column',
                attrs: null,
                innerBlocks: [
                    {
                        blockName: 'core/paragraph',
                        attrs: null,
                        innerBlocks: [],
                        innerHTML: '\n<p>Left</p>\n',
                    },
                ],
                innerHTML: '\n<div class="wp-block-column"></div>\n',
            },
            {

```

```

        blockName: 'core/column',
        attrs: null,
        innerBlocks: [
            {
                blockName: 'core/paragraph',
                attrs: null,
                innerBlocks: [],
                innerHTML: '\n<p><strong>Middle</strong></p>\n',
            },
        ],
        innerHTML: '\n<div class="wp-block-column"></div>\n',
    },
    {
        blockName: 'core/column',
        attrs: null,
        innerBlocks: [],
        innerHTML: '\n<div class="wp-block-column"></div>\n',
    },
],
innerHTML:
    '\n<div class="wp-block-columns has-3-columns">\n\n\n</d
},
];

```

Parameters

- `doc string`: The HTML document to parse.

Returns

- `ParsedBlock []`: A block-based representation of the input HTML.

Theory

What is different about this one from the spec-parser?

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. It minimizes data copying and passing through the use of globals for tracking state through the parse. Between every token (a block comment delimiter) we can instrument the parser and intervene should we want to; for example we might put a hard limit on how long we can be parsing a document or provide additional debugging diagnostics for a document.

The spec parser is defined via a *Parsing Expression Grammar* (PEG) which answers many questions inherently that we must answer explicitly in this parser. The goal for this implementation is to match the characteristics of the PEG so that it can be directly swapped out and so that the only changes are better runtime performance and memory usage.

How does it work?

Every serialized Gutenberg document is nominally an HTML document which, in addition to normal HTML, may also contain specially designed HTML comments — the block comment delimiters — which separate and isolate the blocks serialized in the document.

This parser attempts to create a state-machine around the transitions triggered from those delimiters — the “tokens” of the grammar. Every time we find one we should only be doing either of:

- enter a new block;
- exit out of a block.

Those actions have different effects depending on the context; for instance, when we exit a block we either need to add it to the output block list *or* we need to append it as the next `innerBlock` on the parent block below it in the block stack (the place where we track open blocks). The details are documented below.

The biggest challenge in this parser is making the right accounting of indices required to construct the `innerHTML` values for each block at every level of nesting depth. We take a simple approach:

- Start each newly opened block with an empty `innerHTML`.
- Whenever we push a first block into the `innerBlocks` list, add the content from where the content of the parent block started to where this inner block starts.
- Whenever we push another block into the `innerBlocks` list, add the content from where the previous inner block ended to where this inner block starts.
- When we close out an open block, add the content from where the last inner block ended to where the closing block delimiter starts.
- If there are no inner blocks then we take the entire content between the opening and closing block comment delimiters as the `innerHTML`.

I meant, how does it perform?

This parser operates much faster than the generated parser from the specification. Because we know more about the parsing than the PEG does we can take advantage of several tricks to improve our speed and memory usage:

- We only have one or two distinct tokens, depending on how you look at it, and they are all readily matched via a regular expression. Instead of parsing on a character-per-character basis we can allow the PCRE RegExp engine to skip over large swaths of the document for us in order to find those tokens.
- Since `preg_match()` takes an `offset` parameter we can crawl through the input without passing copies of the input text on every step. We can track our position in the string and only pass a number instead.
- Not copying all those strings means that we’ll also skip many memory allocations.

Further, tokenizing with a RegExp brings an additional advantage. The parser generated by the PEG provides predictable performance characteristics in exchange for control over tokenization rules — it doesn’t allow us to define RegExp patterns in the rules so as to guard against *e.g.* cataclysmic backtracking that would break the PEG guarantees.

However, since our “token language” of the block comment delimiters is *regular* and *can* be trivially matched with RegExp patterns, we can do that here and then something magical happens: we jump out of PHP or JavaScript and into a highly-optimized RegExp engine written in C or C++ on the host system. We thereby leave the virtual machine and its overhead.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/block-serialization-default-parser”](#)

[Previous @wordpress/block-library](#) [Previous: @wordpress/block-library](#)

[Next @wordpress/block-serialization-spec-parser](#) [Next: @wordpress/block-serialization-spec-parser](#)

@wordpress/block-serialization-spec-parser

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This library contains the grammar file (`grammar.pegjs`) for WordPress posts which is a block serialization *specification* which is used to generate the actual *parser* which is also bundled in this package.

PEG parser generators are available in many languages, though different libraries may require some translation of this grammar into their syntax. For more information see:

- [PEG.js](#)
- [Parsing expression grammar](#)

Installation

Install the module

```
npm install @wordpress/block-serialization-spec-parser --save
```

Usage

```
import { parse } from '@wordpress/block-serialization-spec-parser';

parse( '<!-- wp:core/more --><!--more--><!-- /wp:core/more -->' );
// [ {"attrs": null, "blockName": "core/more", "innerBlocks": [], "innerHTML": "" } ]
```

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/block-serialization-spec-parser](#)

[Previous: @wordpress/block-serialization-default-parser](#) [Previous: @wordpress/block-serialization-default-parser](#)
[Next: @wordpress/blocks](#) [Next: @wordpress/blocks](#)

@wordpress/blocks

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [cloneBlock](#)
 - [createBlock](#)
 - [createBlocksFromInnerBlocksTemplate](#)
 - [doBlocksMatchTemplate](#)
 - [findTransform](#)
 - [getBlockAttributes](#)
 - [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](#)
- [rawHandler](#)
- [registerBlockCollection](#)
- [registerBlockStyle](#)
- [registerBlockType](#)
- [registerBlockVariation](#)
- [serialize](#)
- [serializeRawBlock](#)
- [setCategories](#)
- [setDefaultBlockName](#)
- [setFreeformContentHandlerName](#)
- [setGroupingBlockName](#)
- [setUnregisteredTypeHandlerName](#)
- [store](#)
- [switchToBlockType](#)
- [synchronizeBlocksWithTemplate](#)
- [unregisterBlockStyle](#)
- [unregisterBlockType](#)
- [unregisterBlockVariation](#)
- [updateCategory](#)
- [validateBlock](#)
- [withBlockContentContext](#)

- [Contributing to this package](#)

[↑ Back to top](#)

“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 Object`: Block instance.
- `mergeAttributes Object`: Block attributes.
- `newInnerBlocks ?Array`: Nested blocks.

Returns

- `Object`: A cloned block.

createBlock

Returns a block object given its type and attributes.

Parameters

- `name string`: Block name.
- `attributes Object`: Block attributes.
- `innerBlocks ?Array`: Nested blocks.

Returns

- `Object`: 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*: Nested blocks or InnerBlocks templates.

Returns

- *Object []*: Array of Block objects.

[doBlocksMatchTemplate](#)

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

Parameters

- *blocks Array*: Block list.
- *template Array*: 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 Object []*: Transforms to search.
- *predicate Function*: Function returning true on matching transform.

Returns

- *?Object*: Highest-priority transform candidate.

[getBlockAttributes](#)

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

Parameters

- *blockTypeOrName string|Object*: Block type or name.
- *innerHTML string|Node*: Raw block content.
- *attributes ?Object*: Known block attributes (from delimiters).

Returns

- `Object`: All block attributes.

[getBlockContent](#)

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

Parameters

- `block Object`: 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](#)

Create a block object from the example API.

Parameters

- `name string`:
- `example Object`:

Returns

- `Object`: block.

[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|Object`): Block name or type object
- *feature* `string`: Feature to retrieve
- *defaultSupports* `*`: Default value to return if not explicitly defined

Returns

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

Returns

- `Array`: Block transforms for direction.

[getBlockType](#)

Returns a registered block type.

Parameters

- *name* `string`: Block name.

Returns

- `?Object`: Block type.

[getBlockTypes](#)

Returns all registered blocks.

Returns

- `Array`: Block settings.

[getChildBlockNames](#)

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

Parameters

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

Returns

- `Array`: Array of child block names.

[getDefaultValue](#)

Retrieves the default block name.

Returns

- `?string`: Block name.

[getFreeformContentHandlerName](#)

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

Returns

- `?string`: Block name.

[getGroupingBlockName](#)

Retrieves name of block used for handling grouping interactions.

Returns

- `?string`: 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* `Array`: Blocks array.

Returns

- `Array`: 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|Object`: Block type or name.
- *attributes* `Object`: Block attributes.
- *innerBlocks* `?Array`: 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|Object`: Block type or name.
- *attributes* `Object`: Block attributes.
- *innerBlocks* `?Array`: Nested blocks.

Returns

- `Object|string`: 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`: Block name.

[hasBlockSupport](#)

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

Parameters

- *nameOrType* (`string|Object`): Block name or type object.
- *feature* `string`: Feature to test.
- *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* `Object`: 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* `Object`: 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* `WPBlock`: Block Object

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 WPBlock`: Block Object

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 string|Object`: Block type.
- `attributes Object`: 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 *`: 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 one possible icon structure in the codebase.

Parameters

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

Returns

- `WPBlockTypeIconDescriptor`: 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

- `Array`: 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`: Block's raw content.
- *attributeSchema* `Object`: Attribute's schema.

Returns

- *: Attribute value.

[pasteHandler](#)

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

Parameters

- *options Object*:
- *options.HTML [string]*: The HTML to convert.
- *options.plainText [string]*: Plain text version.
- *options.mode [string]*: 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 [Array]*: The tag into which content will be inserted.

Returns

- *Array | string*: A list of blocks or a string, depending on `handlerMode`.

[rawHandler](#)

Converts an HTML string to known blocks.

Parameters

- *\$1 Object*:
- *\$1.HTML string*: The HTML to convert.

Returns

- *Array*: A list of blocks.

[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/block-editor';

// 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 Object*: The block collection settings.
- *settings.title string*: The title to display in the block inserter.

- *settings.icon* [Object]: The icon to display in the block inserter.

registerBlockStyle

Registers a new block style for the given block.

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

- *blockName* string: Name of block (example: “core/latest-posts”).
- *styleVariation* Object: 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 | Object`: Block type name or its metadata.
- *settings* `Object`: Block settings.

Returns

- `WPBlockType | 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* `WPBlockVariation`: Object describing a block variation.

[serialize](#)

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

Parameters

- *blocks* `Array`: Block(s) to serialize.
- *options* `WPBlockSerializationOptions`: 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` `WPRawBlock`: A block node as returned by a valid parser.
- `options` [Options]: 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* WPBlockCategory []: 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#creatReduxStore>

Type

- Object

[switchToBlockType](#)

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

Parameters

- *blocks* Array|Object: Blocks array or block object.
- *name* string: Block name.

Returns

- ?Array: 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* Array: Block list.
- *template* Array: Block template.

Returns

- **Array**: Updated Block list.

[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

- `WPBlockType | 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`: 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* `WPBlockCategory`: 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* `WPBlock`: block object.
- *blockTypeOrName* [`WPBlockType` | `string`]: Block type or name, inferred from block if not given.

Returns

- `[boolean, Array<LoggerItem>]`: validation results.

[withBlockContentContext](#)

Deprecated

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

Parameters

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

Returns

- `Component`: 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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/blocks”](#)

[Previous @wordpress/block-serialization-spec-parser](#) [Previous: @wordpress/block-serialization-spec-parser](#)

[Next @wordpress/browserslist-config](#) [Next: @wordpress/browserslist-config](#)

@wordpress/browserslist-config

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

[WordPress Browserslist](#) shareable config for [Browserslist](#).

[Installation](#)

Install the module

```
$ npm install browserslist @wordpress/browserslist-config --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[Usage](#)

Add this to your `package.json` file:

```
"browserslist": [  
    "extends @wordpress/browserslist-config"  
]
```

Alternatively, add this to `.browserslistrc` file:

```
extends @wordpress/browserslist-config
```

This package when imported returns an array of supported browsers, for more configuration examples including Autoprefixer, Babel, ESLint, PostCSS, and stylelint see the [Browserslist examples](#) repo.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/browserslist-config”](#)

[Previous @wordpress/blocks](#) [Previous: @wordpress/blocks](#)
[Next @wordpress/commands](#) [Next: @wordpress/commands](#)

@wordpress/commands

In this article

Table of Contents

- [Types of commands](#)
 - [Static commands](#)
 - [Dynamic commands](#)
- [Contextual commands](#)
- [WordPress Data API](#)
- [Installation](#)
- [API](#)
 - [store](#)
 - [useCommand](#)
 - [useCommandLoader](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Commands is a generic package that allows registering and modifying commands to be displayed using the commands menu, also called the Command Palette. The Command Palette can be accessed in the Editor using `cmd+k`.

Types of commands

There are two ways to register commands: static or dynamic. Both methods receive a command object as an argument, which provides:

- `name`: A unique machine-readable name for the command
- `label`: A human-readable label
- `icon`: An SVG icon
- `callback`: A callback function that is called when the command is selected
- `context`: (Optional) The context of the command

Static commands

Static commands can be registered using the `wp.data.dispatch(wp.commands.store).registerCommand` action or using the `wp.commands.useCommand` React hook. Static commands are commonly used to perform a specific action. These could include adding a new page or opening a section of the Editor interface, such as opening the Editor Preferences modal. See the `useCommand` [code example](#) below.

Dynamic commands

Dynamic commands, on the other hand, are registered using “command loaders”, `wp.commands.useCommandLoader`. These loaders are needed when the command list depends on a search term entered by the user in the Command Palette input or when some commands are only available when some conditions are met.

For example, when a user types “contact”, the Command Palette needs to filter the available pages using that input to try and find the Contact page. See the `useCommandLoader` [code example](#) below.

Contextual commands

Static and dynamic commands can be contextual. This means that in a given context (for example, when navigating the Site Editor or editing a template), some specific commands are given more priority and are visible as soon as you open the Command Palette. Also, when typing the Command Palette, these contextual commands are shown above the rest of the commands.

At the moment, two contexts have been implemented:

- `site-editor`: This is the context that is set when you are navigating in the site editor (sidebar visible).
- `site-editor-edit`: This is the context that is set when you are editing a document (template, template part or page) in the site editor.
As the usage of the Command Palette expands, more contexts will be added.

Attaching a command or command loader to a given context is as simple as adding the `context` property (with the right context value from the available contexts above) to the `useCommand` or `useCommandLoader` calls.

WordPress Data API

The Command Palette also offers a number of [selectors and actions](#) to manipulate its state, which include:

- Retrieving the registered commands and command loaders using the following selectors
`getCommands` and `getCommandLoader`
- Checking if the Command Palette is open using the `isOpen` selector.
- Programmatically open or close the Command Palette using the `open` and `close` actions.

See the [Commands Data](#) documentation for more information.

Installation

Install the module

```
npm install @wordpress/commands --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

store

Store definition for the commands namespace.

Related

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

Usage

```
import { store as commandsStore } from '@wordpress/commands';
import { useDispatch } from '@wordpress/data';
...
const { open: openCommandCenter } = useDispatch( commandsStore );
```

Type

- `Object`

useCommand

Attach a command to the command palette. Used for static commands.

Usage

```
import { useCommand } from '@wordpress/commands';
import { plus } from '@wordpress/icons';
```

```

useCommand( {
    name: 'myplugin/my-command-name',
    label: __( 'Add new post' ),
    icon: plus,
    callback: ( { close } ) => {
        document.location.href = 'post-new.php';
        close();
    },
} );

```

Parameters

- *command* import('../store/actions').WPCommandConfig: command config.

[useCommandLoader](#)

Attach a command loader to the command palette. Used for dynamic commands.

Usage

```

import { useCommandLoader } from '@wordpress/commands';
import { post, page, layout, symbolFilled } from '@wordpress/icons';

const icons = {
    post,
    page,
    wp_template: layout,
    wp_template_part: symbolFilled,
};

function usePageSearchCommandLoader( { search } ) {
    // Retrieve the pages for the "search" term.
    const { records, isLoading } = useSelect( ( select ) => {
        const { getEntityRecords } = select( coreStore );
        const query = {
            search: !! search ? search : undefined,
            per_page: 10,
            orderby: search ? 'relevance' : 'date',
        };
        return {
            records: getEntityRecords( 'postType', 'page', query ),
            isLoading: ! select( coreStore ).hasFinishedResolution(
                'getEntityRecords',
                'postType', 'page', query
            ),
        };
    }, [ search ] );
}

// Create the commands.
const commands = useMemo( () => {
    return ( records ?? [] ).slice( 0, 10 ).map( ( record ) => {
        return {
            name: record.title?.rendered + ' ' + record.id,
        };
    });
});

```

```

        label: record.title?.rendered
            ? record.title?.rendered
            : __( '(no title)' ),
        icon: icons[ postType ],
        callback: ( { close } ) => {
            const args = {
                postType,
                postId: record.id,
                ...extraArgs,
            };
            document.location = addQueryArgs( 'site-editor.php', a
                close();
            },
        );
    );
},
[ records, history ] );

return {
    commands,
    isLoading,
};
}

useCommandLoader( {
    name: 'myplugin/page-search',
    hook: usePageSearchCommandLoader,
} );

```

Parameters

- *loader* import('.../store/actions').WPCommandLoaderConfig: command loader config.

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](#).

First published

April 6, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/commands](#)

@wordpress/components

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
 - [Popovers](#)
- [Docs & examples](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This package includes a library of generic WordPress components to be used for creating common UI elements shared between screens and features of the WordPress dashboard.

[Installation](#)

```
npm install @wordpress/components --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.*

[Usage](#)

Within Gutenberg, these components can be accessed by importing from the `components` root directory:

```
/**  
 * WordPress dependencies  
 */  
import { Button } from '@wordpress/components';  
  
export default function MyButton() {  
    return <Button>Click Me!</Button>;  
}
```

Many components include CSS to add styles, which you will need to load in order for them to appear correctly. Within WordPress, add the `wp-components` stylesheet as a dependency of your plugin's stylesheet. See [wp_enqueue_style documentation](#) for how to specify dependencies.

In non-WordPress projects, link to the `build-style/style.css` file directly, it is located at `node_modules/@wordpress/components/build-style/style.css`.

[Popovers](#)

By default, the `Popover` component will render within an extra element appended to the body of the document.

If you want to precisely control where the popovers render, you will need to use the `Popover.Slot` component.

The following example illustrates how you can wrap a component using a `Popover` and have those popovers render to a single location in the DOM.

```
/**  
 * External dependencies  
 */  
import { Popover, SlotFillProvider } from '@wordpress/components';  
  
/**  
 * Internal dependencies  
 */  
import { MyComponentWithPopover } from './my-component';  
  
const Example = () => {  
    <SlotFillProvider>  
        <MyComponentWithPopover />  
        <Popover.Slot />  
    </SlotFillProvider>;  
};
```

[Docs & examples](#)

You can browse the components docs and examples at <https://wordpress.github.io/gutenberg/>

[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](#).

This package also has its own [contributing information](#) where you can find additional details.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/components”](#)

[Previous @wordpress/commands](#) [Previous: @wordpress/commands](#)

[Next @wordpress/compose](#) [Next: @wordpress/compose](#)

@wordpress/compose

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [compose](#)
 - [createHigherOrderComponent](#)
 - [debounce](#)
 - [ifCondition](#)
 - [pipe](#)
 - [pure](#)
 - [throttle](#)
 - [useAsyncList](#)
 - [useConstrainedTabbing](#)
 - [useCopyOnClick](#)
 - [useCopyToClipboard](#)
 - [useDebounce](#)
 - [useDebouncedInput](#)
 - [useDisabled](#)
 - [useFocusableIframe](#)
 - [useFocusOnMount](#)
 - [useFocusReturn](#)
 - [useInstanceId](#)
 - [useIsomorphicLayoutEffect](#)
 - [useKeyboardShortcut](#)
 - [useMediaQuery](#)
 - [useMergeRefs](#)
 - [usePrevious](#)
 - [useReducedMotion](#)
 - [useRefEffect](#)
 - [useResizeObserver](#)
 - [useStateWithHistory](#)
 - [useThrottle](#)
 - [useViewportMatch](#)
 - [useWarnOnChange](#)
 - [withGlobalEvents](#)
 - [withInstanceId](#)
 - [withSafeTimeout](#)
 - [withState](#)
- [Contributing to this package](#)

[↑ Back to top](#)

The `compose` package is a collection of handy [Hooks](#) and [Higher Order Components](#) (HOCs) you can use to wrap your WordPress components and provide some basic features like: state, instance id, pure...

The `compose` function is inspired by [flowRight](#) from Lodash and works the same way. It comes from functional programming, and allows you to compose any number of functions. You might also think of this as layering functions; `compose` will execute the last function first, then sequentially move back through the previous functions passing the result of each function upward.

An example that illustrates it for two functions:

```
const compose = ( f, g ) => x
  => f( g( x ) );
```

Here's a simplified example of `compose` in use from Gutenberg's [PluginSidebar component](#):

Using `compose`:

```
const applyWithSelect = withSelect( ( select, ownProps ) => {
    return doSomething( select, ownProps );
} );
const applyWithDispatch = withDispatch( ( dispatch, ownProps ) => {
    return doSomethingElse( dispatch, ownProps );
} );

export default compose(
    withPluginContext,
    applyWithSelect,
    applyWithDispatch
)( PluginSidebarMoreMenuItem );
```

Without `compose`, the code would look like this:

```
const applyWithSelect = withSelect( ( select, ownProps ) => {
    return doSomething( select, ownProps );
} );
const applyWithDispatch = withDispatch( ( dispatch, ownProps ) => {
    return doSomethingElse( dispatch, ownProps );
} );

export default withPluginContext(
    applyWithSelect( applyWithDispatch( PluginSidebarMoreMenuItem ) )
);
```

Installation

Install the module

```
npm install @wordpress/compose --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

For more details, you can refer to each Higher Order Component's README file. [Available components are located here.](#)

compose

Composes multiple higher-order components into a single higher-order component. Performs right-to-left function composition, where each successive invocation is supplied the return value of the previous.

This is inspired by lodash's `flowRight` function.

Related

- <https://docs-lodash.com/v4/flow-right/>

createHigherOrderComponent

Given a function mapping a component to an enhanced component and modifier name, returns the enhanced component augmented with a generated displayName.

Parameters

- `mapComponent (Inner: TInner) => TOuter`: Function mapping component to enhanced component.
- `modifierName string`: Seed name from which to generate display name.

Returns

- Component class with generated display name assigned.

debounce

A simplified and properly typed version of lodash's `debounce`, that always uses timers instead of sometimes using rAF.

Creates a debounced function that delays invoking `func` until after `wait` milliseconds have elapsed since the last time the debounced function was invoked. The debounced function comes with a `cancel` method to cancel delayed `func` invocations and a `flush` method to immediately invoke them. Provide `options` to indicate whether `func` should be invoked on the leading and/or trailing edge of the `wait` timeout. The `func` is invoked with the last arguments provided to the debounced function. Subsequent calls to the debounced function return the result of the last `func` invocation.

Note: If `leading` and `trailing` options are `true`, `func` is invoked on the trailing edge of the timeout only if the debounced function is invoked more than once during the `wait` timeout.

If `wait` is `0` and `leading` is `false`, `func` invocation is deferred until the next tick, similar to `setTimeout` with a timeout of `0`.

Parameters

- `func Function`: The function to debounce.

- *wait* `number`: The number of milliseconds to delay.
- *options* `Partial< DebounceOptions >`: The options object.
- *options.leading* `boolean`: Specify invoking on the leading edge of the timeout.
- *options.maxWait* `number`: The maximum time `func` is allowed to be delayed before it's invoked.
- *options.trailing* `boolean`: Specify invoking on the trailing edge of the timeout.

Returns

- Returns the new debounced function.

ifCondition

Higher-order component creator, creating a new component which renders if the given condition is satisfied or with the given optional prop name.

Usage

```
type Props = { foo: string };
const Component = ( props: Props ) => <div>{ props.foo }</div>;
const ConditionalComponent = ifCondition( ( props: Props ) => props.foo.length === 0 )
<ConditionalComponent foo="" />; // => null
<ConditionalComponent foo="bar" />; // => <div>bar</div>;
```

Parameters

- *predicate* (`props: Props`) => `boolean`: Function to test condition.

Returns

- Higher-order component.

pipe

Composes multiple higher-order components into a single higher-order component. Performs left-to-right function composition, where each successive invocation is supplied the return value of the previous.

This is inspired by lodash's `flow` function.

Related

- <https://docs-lodash.com/v4/flow/>

pure

Deprecated Use `memo` or `PureComponent` instead.

Given a component returns the enhanced component augmented with a component only re-rendering when its props/state change

throttle

A simplified and properly typed version of lodash's `throttle`, that always uses timers instead of sometimes using rAF.

Creates a throttled function that only invokes `func` at most once per every `wait` milliseconds. The throttled function comes with a `cancel` method to cancel delayed `func` invocations and a `flush` method to immediately invoke them. Provide `options` to indicate whether `func` should be invoked on the leading and/or trailing edge of the `wait` timeout. The `func` is invoked with the last arguments provided to the throttled function. Subsequent calls to the throttled function return the result of the last `func` invocation.

Note: If `leading` and `trailing` options are `true`, `func` is invoked on the trailing edge of the timeout only if the throttled function is invoked more than once during the `wait` timeout.

If `wait` is `0` and `leading` is `false`, `func` invocation is deferred until the next tick, similar to `setTimeout` with a timeout of `0`.

Parameters

- `func` Function: The function to throttle.
- `wait` number: The number of milliseconds to throttle invocations to.
- `options` Partial< ThrottleOptions >: The options object.
- `options.leading` boolean: Specify invoking on the leading edge of the timeout.
- `options.trailing` boolean: Specify invoking on the trailing edge of the timeout.

Returns

- Returns the new throttled function.

useAsyncList

React hook returns an array which items get asynchronously appended from a source array. This behavior is useful if we want to render a list of items asynchronously for performance reasons.

Parameters

- `list` T[]: Source array.
- `config` AsyncListConfig: Configuration object.

Returns

- T[]: Async array.

useConstrainedTabbing

In Dialogs/modals, the tabbing must be constrained to the content of the wrapper element. This hook adds the behavior to the returned ref.

Usage

```
import { useConstrainedTabbing } from '@wordpress/compose';

const ConstrainedTabbingExample = () => {
    const constrainedTabbingRef = useConstrainedTabbing();
```

```
        return (
          <div ref={ constrainedTabbingRef }>
            <Button />
            <Button />
          </div>
        );
    };
}
```

Returns

- `import('react').RefCallback<Element>: Element Ref.`

[useCopyOnClick](#)

Deprecated

Copies the text to the clipboard when the element is clicked.

Parameters

- `ref import('react').RefObject<string | Element | NodeListOf<Element>>: Reference with the element.`
- `text string | Function: The text to copy.`
- `timeout [number]: Optional timeout to reset the returned state. 4 seconds by default.`

Returns

- `boolean: Whether or not the text has been copied. Resets after the timeout.`

[useCopyToClipboard](#)

Copies the given text to the clipboard when the element is clicked.

Parameters

- `text string | (() => string): The text to copy. Use a function if not already available and expensive to compute.`
- `onSuccess Function: Called when the text is copied.`

Returns

- `import('react').Ref<TElementType>: A ref to assign to the target element.`

[useDebounce](#)

Debounces a function similar to Lodash's `debounce`. A new debounced function will be returned and any scheduled calls cancelled if any of the arguments change, including the function to debounce, so please wrap functions created on render in components in `useCallback`.

Related

- <https://docs-lodash.com/v4/debounce/>

Parameters

- *fn TFunc*: The function to debounce.
- *wait [number]*: The number of milliseconds to delay.
- *options [import('.../utils/debounce').DebounceOptions]*: The options object.

Returns

- `import('.../utils/debounce').DebouncedFunc<TFunc>`: Debounced function.

[useDebouncedInput](#)

Helper hook for input fields that need to debounce the value before using it.

Parameters

- *defaultValue any*: The default value to use.

Returns

- `[string, Function, string]`: The input value, the setter and the debounced input value.

[useDisabled](#)

In some circumstances, such as block previews, all focusable DOM elements (input fields, links, buttons, etc.) need to be disabled. This hook adds the behavior to disable nested DOM elements to the returned ref.

If you can, prefer the use of the inert HTML attribute.

Usage

```
import { useDisabled } from '@wordpress/compose';

const DisabledExample = () => {
    const disabledRef = useDisabled();
    return (
        <div ref={ disabledRef }>
            <a href="#">This link will have tabindex set to -1</a>
            <input
                placeholder="This input will have the disabled attribute a"
                type="text"
            />
        </div>
    );
}
```

Parameters

- *config Object*: Configuration object.
- *config.isDisabled boolean*=: Whether the element should be disabled.

Returns

- `import('react').RefCallback<HTMLElement>: Element Ref.`

[useFocusableIframe](#)

Dispatches a bubbling focus event when the iframe receives focus. Use `onFocus` as usual on the iframe or a parent element.

Returns

- `RefCallback< HTMLIFrameElement >: Ref to pass to the iframe.`

[useFocusOnMount](#)

Hook used to focus the first tabbable element on mount.

Usage

```
import { useFocusOnMount } from '@wordpress/compose';

const WithFocusOnMount = () => {
    const ref = useFocusOnMount();
    return (
        <div ref={ ref }>
            <Button />
            <Button />
        </div>
    );
}
```

Parameters

- `focusOnMount boolean | 'firstElement': Focus on mount mode.`

Returns

- `import('react').RefCallback<HTMLElement>: Ref callback.`

[useFocusReturn](#)

Adds the unmount behavior of returning focus to the element which had it previously as is expected for roles like menus or dialogs.

Usage

```
import { useFocusReturn } from '@wordpress/compose';

const WithFocusReturn = () => {
    const ref = useFocusReturn();
    return (
        <div ref={ ref }>
            <Button />
            <Button />
    
```

```
        </div>
    );
}
```

Parameters

- *onFocusReturn* [() => void]: Overrides the default return behavior.

Returns

- import('react').RefCallback<HTMLElement>: Element Ref.

[useInstanceId](#)

Provides a unique instance ID.

Parameters

- *object object*: Object reference to create an id for.
- *prefix* [string]: Prefix for the unique id.
- *preferredId* [string | number]: Default ID to use.

Returns

- string | number: The unique instance id.

[useIsomorphicLayoutEffect](#)

Preferred over direct usage of `useLayoutEffect` when supporting server rendered components (SSR) because currently React throws a warning when using `useLayoutEffect` in that environment.

[useKeyboardShortcut](#)

Attach a keyboard shortcut handler.

Related

- <https://craig.is/killing/mice#api.bind> for information about the `callback` parameter.

Parameters

- *shortcuts* string[] | string: Keyboard Shortcuts.
- *callback* (e: import('mousetrap').ExtendedKeyboardEvent, combo: string) => void: Shortcut callback.
- *options* WPKeyboardShortcutConfig: Shortcut options.

[useMediaQuery](#)

Runs a media query and returns its value when it changes.

Parameters

- *query* [string]: Media Query.

Returns

- `boolean`: return value of the media query.

[useMergeRefs](#)

Merges refs into one ref callback.

It also ensures that the merged ref callbacks are only called when they change (as a result of a `useCallback` dependency update) OR when the ref value changes, just as React does when passing a single ref callback to the component.

As expected, if you pass a new function on every render, the ref callback will be called after every render.

If you don't wish a ref callback to be called after every render, wrap it with `useCallback(callback, dependencies)`. When a dependency changes, the old ref callback will be called with `null` and the new ref callback will be called with the same value.

To make ref callbacks easier to use, you can also pass the result of `useRefEffect`, which makes cleanup easier by allowing you to return a cleanup function instead of handling `null`.

It's also possible to *disable* a ref (and its behaviour) by simply not passing the ref.

```
const ref = useRefEffect( ( node ) => {
  node.addEventListener( ... );
  return () => {
    node.removeEventListener( ... );
  };
}, [ ...dependencies ] );
const otherRef = useRef();
const mergedRefs = useMergeRefs( [
  enabled && ref,
  otherRef,
] );
return <div ref={ mergedRefs } />;
```

Parameters

- `refs Array<TRef>`: The refs to be merged.

Returns

- `import('react').RefCallback<TypeFromRef<TRef>>`: The merged ref callback.

[usePrevious](#)

Use something's value from the previous render. Based on <https://usehooks.com/usePrevious/>.

Parameters

- `value T`: The value to track.

Returns

- `T | undefined`: The value from the previous render.

[useReducedMotion](#)

Hook returning whether the user has a preference for reduced motion.

Returns

- `boolean`: Reduced motion preference value.

[useRefEffect](#)

Effect-like ref callback. Just like with `useEffect`, this allows you to return a cleanup function to be run if the ref changes or one of the dependencies changes. The ref is provided as an argument to the callback functions. The main difference between this and `useEffect` is that the `useEffect` callback is not called when the ref changes, but this is. Pass the returned ref callback as the component's ref and merge multiple refs with `useMergeRefs`.

It's worth noting that if the dependencies array is empty, there's not strictly a need to clean up event handlers for example, because the node is to be removed. It *is* necessary if you add dependencies because the ref callback will be called multiple times for the same node.

Parameters

- `callback (node: TElement) => (() => void) | void`: Callback with ref as argument.
- `dependencies DependencyList`: Dependencies of the callback.

Returns

- `RefCallback< TElement | null >`: Ref callback.

[useResizeObserver](#)

Hook which allows to listen the resize event of any target element when it changes sizes. Note: `useResizeObserver` will report `null` until after first render.

Usage

```
const App = () => {
  const [ resizeListener, sizes ] = useResizeObserver();

  return (
    <div>
      { resizeListener }
      Your content here
    </div>
  );
};
```

[useStateWithHistory](#)

useState with undo/redo history.

Parameters

- *initialValue* T: Initial value.

Returns

- Value, setValue, hasUndo, hasRedo, undo, redo.

[useThrottle](#)

Throttles a function similar to Lodash's `throttle`. A new throttled function will be returned and any scheduled calls cancelled if any of the arguments change, including the function to throttle, so please wrap functions created on render in components in `useCallback`.

Related

- <https://docs-lodash.com/v4/throttle/>

Parameters

- *fn* TFunc: The function to throttle.
- *wait* [number]: The number of milliseconds to throttle invocations to.
- *options* [import('.../..../utils/throttle').ThrottleOptions]: The options object. See linked documentation for details.

Returns

- import('.../..../utils/debounce').DebouncedFunc<TFunc>: Throttled function.

[useViewportMatch](#)

Returns true if the viewport matches the given query, or false otherwise.

Usage

```
useViewportMatch( 'huge' , '<' );
useViewportMatch( 'medium' );
```

Parameters

- *breakpoint* WPBreakpoint: Breakpoint size name.
- *operator* [WPViewportOperator]: Viewport operator.

Returns

- boolean: Whether viewport matches query.

[useWarnOnChange](#)

Hook that performs a shallow comparison between the previous value of an object and the new one, if there's a difference, it prints it to the console. This is useful in performance related work, to check why a component re-renders.

Usage

```
function MyComponent( props ) {
  useWarnOnChange( props );

  return 'Something';
}
```

Parameters

- *object* **Object**: Object which changes to compare.
- *prefix* **string**: Just a prefix to show when console logging.

[withGlobalEvents](#)

Deprecated

Higher-order component creator which, given an object of DOM event types and values corresponding to a callback function name on the component, will create or update a window event handler to invoke the callback when an event occurs. On behalf of the consuming developer, the higher-order component manages unbinding when the component unmounts, and binding at most a single event handler for the entire application.

Parameters

- *eventTypesToHandlers* **Record<keyof GlobalEventHandlersEventMap, string>**: Object with keys of DOM event type, the value a name of the function on the original component's instance which handles the event.

Returns

- **any**: Higher-order component.

[withInstanceId](#)

A Higher Order Component used to provide a unique instance ID by component.

[withSafeTimeout](#)

A higher-order component used to provide and manage delayed function calls that ought to be bound to a component's lifecycle.

[withState](#)

Deprecated Use `useState` instead.

A Higher Order Component used to provide and manage internal component state via props.

Parameters

- `initialState` `any`: Optional initial state of the component.

Returns

- `any`: A higher order component wrapper accepting a component that takes the state props + its own props + `setState` and returning a component that only accepts the own props.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/compose”](#)

[Previous @wordpress/components](#) [Previous: @wordpress/components](#)
[Next @wordpress/core-commands](#) [Next: @wordpress/core-commands](#)

@wordpress/core-commands

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [privateApis](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This package includes a list of reusable WordPress Admin commands. These commands can be used in multiple WP Admin pages.

Installation

Install the module

```
npm install @wordpress/core-commands --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

privateApis

Undocumented declaration.

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](#).

First published

April 27, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/core-commands](#)

[Previous @wordpress/compose](#) [Previous: @wordpress/compose](#)

[Next @wordpress/core-data](#) [Next: @wordpress/core-data](#)

@wordpress/core-data

In this article

Table of Contents

- [Installation](#)
- [Example](#)
- [What's an entity?](#)
 - [Connecting the entity with the data source](#)

- [Interacting with entity records](#)

- [Actions](#)

- [addEntities](#)
- [deleteEntityRecord](#)
- [editEntityRecord](#)
- [receiveDefaultTemplateId](#)
- [receiveEntityRecords](#)
- [receiveNavigationFallbackId](#)
- [receiveRevisions](#)
- [receiveThemeSupports](#)
- [receiveUploadPermissions](#)
- [redo](#)
- [saveEditedEntityRecord](#)
- [saveEntityRecord](#)
- [undo](#)

- [Selectors](#)

- [canUser](#)
- [canUserEditEntityRecord](#)
- [getAuthors](#)
- [getAutosave](#)
- [getAutosaves](#)
- [getBlockPatternCategories](#)
- [getBlockPatterns](#)
- [getCurrentTheme](#)
- [getCurrentThemeGlobalStylesRevisions](#)
- [getCurrentUser](#)
- [getDefaultTemplateId](#)
- [getEditedEntityRecord](#)
- [getEmbedPreview](#)
- [getEntitiesByKind](#)
- [getEntitiesConfig](#)
- [getEntity](#)
- [getEntityConfig](#)
- [getEntityRecord](#)
- [getEntityRecordEdits](#)
- [getEntityRecordNonTransientEdits](#)
- [getEntityRecords](#)
- [getEntityRecordsTotalItems](#)
- [getEntityRecordsTotalPages](#)
- [getLastEntityDeleteError](#)
- [getLastEntitySaveError](#)
- [getRawEntityRecord](#)
- [getRedoEdit](#)
- [getReferenceByDistinctEdits](#)
- [getRevision](#)
- [getRevisions](#)
- [getThemeSupports](#)
- [getUndoEdit](#)
- [getUserPatternCategories](#)
- [getUserQueryResults](#)
- [hasEditsForEntityRecord](#)
- [hasEntityRecords](#)
- [hasFetchedAutosaves](#)
- [hasRedo](#)

- [hasUndo](#)
- [isAutosavingEntityRecord](#)
- [isDeletingEntityRecord](#)
- [isPreviewEmbedFallback](#)
- [isRequestingEmbedPreview](#)
- [isSavingEntityRecord](#)
- [Hooks](#)
 - [useEntityRecord](#)
 - [useEntityRecords](#)
 - [useResourcePermissions](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Core Data is a [data module](#) intended to simplify access to and manipulation of core WordPress entities. It registers its own store and provides a number of selectors which resolve data from the WordPress REST API automatically, along with dispatching action creators to manipulate data. Core data is shipped with [TypeScript definitions for WordPress data types](#).

Used in combination with features of the data module such as [subscribe](#) or [higher-order components](#), it enables a developer to easily add data into the logic and display of their plugin.

Installation

Install the module

```
npm install @wordpress/core-data --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.*

Example

Below is an example of a component which simply renders a list of authors:

```
const { useSelect } = wp.data;

function MyAuthorsListBase() {
    const authors = useSelect( ( select ) => {
        return select( 'core' ).getUsers( { who: 'authors' } );
    }, [] );

    if ( ! authors ) {
        return null;
    }

    return (
        <ul>
            { authors.map( ( author ) => (
                <li key={ author.id }>{ author.name }</li>
            ) ) }
    )
}
```

```
        </ul>
    );
}
```

What's an entity?

An entity represents a data source. Each item within the entity is called an entity record. Available entities are defined in `rootEntitiesConfig` at `./src/entities.js`.

As of right now, the default entities defined by this package map to the [REST API handbook](#), though there is nothing in the design that prevents it from being used to interact with any other API.

What follows is a description of some of the properties of `rootEntitiesConfig`.

Connecting the entity with the data source

baseURL

- Type: `string`.
- Example: `'/wp/v2/users'`.

This property maps the entity to a given endpoint, taking its relative URL as value.

baseURLParams

- Type: `object`.
- Example: `{ context: 'edit' }`.

Additional parameters to the request, added as a query string. Each property will be converted into a field/value pair. For example, given the `baseURL: '/wp/v2/users'` and the `baseURLParams: { context: 'edit' }` the URL would be `/wp/v2/users?context=edit`.

key

- Type: `string`.
- Example: `'slug'`.

The entity engine aims to convert the API response into a number of entity records. Responses can come in different shapes, which are processed differently.

Responses that represent a single object map to a single entity record. For example:

```
{
  "title": "...",
  "description": "...",
  "...": "..."
}
```

Responses that represent a collection shaped as an array, map to as many entity records as elements of the array. For example:

```
[  
  { "id": 1, "name": "...", "...": "..." },  
  { "id": 2, "name": "...", "...": "..." },  
  { "id": 3, "name": "...", "...": "..." }  
]
```

There are also cases in which a response represents a collection shaped as an object, whose key is one of the property's values. Each of the nested objects should be its own entity record. For this case not to be confused with single object/entities, the entity configuration must provide the property key that holds the value acting as the object key. In the following example, the `slug` property's value is acting as the object key, hence the entity config must declare `key: 'slug'` for each nested object to be processed as an individual entity record:

```
{  
  "publish": { "slug": "publish", "name": "Published", "...": "..." },  
  "draft": { "slug": "draft", "name": "Draft", "...": "..." },  
  "future": { "slug": "future", "name": "Future", "...": "..." }  
}
```

[Interacting with entity records](#)

Entity records are unique. For entities that are collections, it's assumed that each record has an `id` property which serves as an identifier to manage it. If the entity defines a `key`, that property would be used as its identifier instead of the assumed `id`.

name

- Type: `string`.
- Example: `user`.

The name of the entity. To be used in the utilities that interact with it (selectors, actions, hooks).

kind

- Type: `string`.
- Example: `root`.

Entities can be grouped by `kind`. To be used in the utilities that interact with them (selectors, actions, hooks).

The package provides general methods to interact with the entities (`getEntityRecords`, `getEntityRecord`, etc.) by leveraging the `kind` and `name` properties:

```
// Get the record collection for the user entity.  
wp.data.select( 'core' ).getEntityRecords( 'root', 'user' );  
  
// Get a single record for the user entity.  
wp.data.select( 'core' ).getEntityRecord( 'root', 'user', recordId );
```

plural

- Type: `string`.
- Example: `statuses`.

In addition to the general utilities (`getEntityRecords`, `getEntityRecord`, etc.), the package dynamically creates nicer-looking methods to interact with the entity records of the root kind, both the collection and single records. Compare the general and nicer-looking methods as follows:

```
// Collection
wp.data.select( 'core' ).getEntityRecords( 'root', 'user' );
wp.data.select( 'core' ).getUsers();

// Single record
wp.data.select( 'core' ).getEntityRecord( 'root', 'user', recordId );
wp.data.select( 'core' ).getUser( recordId );
```

Sometimes, the pluralized form of an entity is not regular (it is not formed by adding a `-s` suffix). The `plural` property of the entity config allows to declare an alternative pluralized form for the dynamic methods created for the entity. For example, given the `status` entity that declares the `statuses` plural, there are the following methods created for it:

```
// Collection
wp.data.select( 'core' ).getstatuses();

// Single record
wp.data.select( 'core' ).getStatus( recordId );
```

[Actions](#)

The following set of dispatching action creators are available on the object returned by `wp.data.dispatch('core')`:

[addEntities](#)

Returns an action object used in adding new entities.

Parameters

- `entities` `Array`: Entities received.

Returns

- `Object`: Action object.

[deleteEntityRecord](#)

Action triggered to delete an entity record.

Parameters

- `kind` `string`: Kind of the deleted entity.
- `name` `string`: Name of the deleted entity.
- `recordId` `string`: Record ID of the deleted entity.
- `query` `?Object`: Special query parameters for the DELETE API call.
- `options` `[Object]`: Delete options.
- `options.__unstableFetch` `[Function]`: Internal use only. Function to call instead of `apiFetch()`. Must return a promise.

- *options.throwOnError* [boolean]: If false, this action suppresses all the exceptions. Defaults to false.

[editEntityRecord](#)

Returns an action object that triggers an edit to an entity record.

Parameters

- *kind* string: Kind of the edited entity record.
- *name* string: Name of the edited entity record.
- *recordId* number|string: Record ID of the edited entity record.
- *edits* Object: The edits.
- *options* Object: Options for the edit.
- *options.undoIgnore* [boolean]: Whether to ignore the edit in undo history or not.

Returns

- Object: Action object.

[receiveDefaultTemplateId](#)

Returns an action object used to set the template for a given query.

Parameters

- *query* Object: The lookup query.
- *templateId* string: The resolved template id.

Returns

- Object: Action object.

[receiveEntityRecords](#)

Returns an action object used in signalling that entity records have been received.

Parameters

- *kind* string: Kind of the received entity record.
- *name* string: Name of the received entity record.
- *records* Array|Object: Records received.
- *query* ?Object: Query Object.
- *invalidateCache* ?boolean: Should invalidate query caches.
- *edits* ?Object: Edits to reset.
- *meta* ?Object: Meta information about pagination.

Returns

- Object: Action object.

[receiveNavigationFallbackId](#)

Returns an action object signalling that the fallback Navigation Menu id has been received.

Parameters

- *fallbackId integer*: the id of the fallback Navigation Menu

Returns

- *Object*: Action object.

[receiveRevisions](#)

Action triggered to receive revision items.

Parameters

- *kind string*: Kind of the received entity record revisions.
- *name string*: Name of the received entity record revisions.
- *recordKey number|string*: The key of the entity record whose revisions you want to fetch.
- *records Array|Object*: Revisions received.
- *query ?Object*: Query Object.
- *invalidateCache ?boolean*: Should invalidate query caches.
- *meta ?Object*: Meta information about pagination.

[receiveThemeSupports](#)

Deprecated since WP 5.9, this is not useful anymore, use the selector directly.

Returns an action object used in signalling that the index has been received.

Returns

- *Object*: Action object.

[receiveUploadPermissions](#)

Deprecated since WP 5.9, use `receiveUserPermission` instead.

Returns an action object used in signalling that Upload permissions have been received.

Parameters

- *hasUploadPermissions boolean*: Does the user have permission to upload files?

Returns

- *Object*: Action object.

[redo](#)

Action triggered to redo the last undoed edit to an entity record, if any.

[saveEditedEntityRecord](#)

Action triggered to save an entity record's edits.

Parameters

- *kind* `string`: Kind of the entity.
- *name* `string`: Name of the entity.
- *recordId* `Object`: ID of the record.
- *options* `Object`: Saving options.

[saveEntityRecord](#)

Action triggered to save an entity record.

Parameters

- *kind* `string`: Kind of the received entity.
- *name* `string`: Name of the received entity.
- *record* `Object`: Record to be saved.
- *options* `Object`: Saving options.
- *options.isAutosave* `[boolean]`: Whether this is an autosave.
- *options._unstableFetch* `[Function]`: Internal use only. Function to call instead of `apiFetch()`. Must return a promise.
- *options.throwOnError* `[boolean]`: If false, this action suppresses all the exceptions. Defaults to false.

[undo](#)

Action triggered to undo the last edit to an entity record, if any.

[Selectors](#)

The following selectors are available on the object returned by `wp.data.select('core')`:

[canUser](#)

Returns whether the current user can perform the given action on the given REST resource.

Calling this may trigger an OPTIONS request to the REST API via the `canUser()` resolver.

<https://developer.wordpress.org/rest-api/reference/>

Parameters

- *state* `State`: Data state.
- *action* `string`: Action to check. One of: ‘create’, ‘read’, ‘update’, ‘delete’.
- *resource* `string`: REST resource to check, e.g. ‘media’ or ‘posts’.
- *id* `EntityRecordKey`: Optional ID of the rest resource to check.

Returns

- `boolean | undefined`: Whether or not the user can perform the action, or `undefined` if the OPTIONS request is still being made.

[canUserEditEntityRecord](#)

Returns whether the current user can edit the given entity.

Calling this may trigger an OPTIONS request to the REST API via the `canUser()` resolver.

<https://developer.wordpress.org/rest-api/reference/>

Parameters

- `state State`: Data state.
- `kind string`: Entity kind.
- `name string`: Entity name.
- `recordId EntityRecordKey`: Record's id.

Returns

- `boolean | undefined`: Whether or not the user can edit, or `undefined` if the OPTIONS request is still being made.

[getAuthors](#)

Deprecated since 11.3. Callers should use

`select('core').getUsers({ who: 'authors' })` instead.

Returns all available authors.

Parameters

- `state State`: Data state.
- `query GetRecordsHttpQuery`: Optional object of query parameters to include with request. For valid query parameters see the [Users page](#) in the REST API Handbook and see the arguments for [List Users](#) and [Retrieve a User](#).

Returns

- `ET.User[]`: Authors list.

[getAutosave](#)

Returns the autosave for the post and author.

Parameters

- `state State`: State tree.
- `postType string`: The type of the parent post.
- `postId EntityRecordKey`: The id of the parent post.
- `authorId EntityRecordKey`: The id of the author.

Returns

- `EntityRecord | undefined`: The autosave for the post and author.

[getAutosaves](#)

Returns the latest autosaves for the post.

May return multiple autosaves since the backend stores one autosave per author for each post.

Parameters

- *state State*: State tree.
- *postType string*: The type of the parent post.
- *postId EntityRecordKey*: The id of the parent post.

Returns

- `Array< any > | undefined`: An array of autosaves for the post, or undefined if there is none.

[getBlockPatternCategories](#)

Retrieve the list of registered block pattern categories.

Parameters

- *state State*: Data state.

Returns

- `Array< any >`: Block pattern category list.

[getBlockPatterns](#)

Retrieve the list of registered block patterns.

Parameters

- *state State*: Data state.

Returns

- `Array< any >`: Block pattern list.

[getCurrentTheme](#)

Return the current theme.

Parameters

- *state State*: Data state.

Returns

- `any`: The current theme.

[getCurrentThemeGlobalStylesRevisions](#)

Deprecated since WordPress 6.5.0. Callers should use `select('core').getRevisions('root', 'globalStyles', ${ recordKey })` instead, where `recordKey` is the id of the global styles parent post.

Returns the revisions of the current global styles theme.

Parameters

- `state State`: Data state.

Returns

- `Array< object > | null`: The current global styles.

[getCurrentUser](#)

Returns the current user.

Parameters

- `state State`: Data state.

Returns

- `undefined< 'edit' >`: Current user object.

[getDefaultTemplateId](#)

Returns the default template use to render a given query.

Parameters

- `state State`: Data state.
- `query TemplateQuery`: Query.

Returns

- `string`: The default template id for the given query.

[getEditedEntityRecord](#)

Returns the specified entity record, merged with its edits.

Parameters

- `state State`: State tree.
- `kind string`: Entity kind.
- `name string`: Entity name.
- `recordId EntityRecordKey`: Record ID.

Returns

- `EntityRecord` | `undefined`: The entity record, merged with its edits.

[getEmbedPreview](#)

Returns the embed preview for the given URL.

Parameters

- `state State`: Data state.
- `url string`: Embedded URL.

Returns

- `any`: Undefined if the preview has not been fetched, otherwise, the preview fetched from the embed preview API.

[getEntitiesByKind](#)

Deprecated since WordPress 6.0. Use `getEntitiesConfig` instead

Returns the loaded entities for the given kind.

Parameters

- `state State`: Data state.
- `kind string`: Entity kind.

Returns

- `Array< any >`: Array of entities with config matching kind.

[getEntitiesConfig](#)

Returns the loaded entities for the given kind.

Parameters

- `state State`: Data state.
- `kind string`: Entity kind.

Returns

- `Array< any >`: Array of entities with config matching kind.

[getEntity](#)

Deprecated since WordPress 6.0. Use `getEntityConfig` instead

Returns the entity config given its kind and name.

Parameters

- *state State*: Data state.
- *kind string*: Entity kind.
- *name string*: Entity name.

Returns

- any: Entity config

[getEntityConfig](#)

Returns the entity config given its kind and name.

Parameters

- *state State*: Data state.
- *kind string*: Entity kind.
- *name string*: Entity name.

Returns

- any: Entity config

[getEntityRecord](#)

Returns the Entity's record object by key. Returns `null` if the value is not yet received, `undefined` if the value entity is known to not exist, or the entity object if it exists and is received.

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *key EntityRecordKey*: Record's key
- *query GetRecordsHttpQuery*: Optional query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available "Retrieve a [Entity kind]".

Returns

- `EntityRecord | undefined: Record`.

[getEntityRecordEdits](#)

Returns the specified entity record's edits.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- `Optional< any >`: The entity record's edits.

[getEntityRecordNonTransientEdits](#)

Returns the specified entity record's non transient edits.

Transient edits don't create an undo level, and are not considered for change detection. They are defined in the entity's config.

Parameters

- `state State`: State tree.
- `kind string`: Entity kind.
- `name string`: Entity name.
- `recordId EntityRecordKey`: Record ID.

Returns

- `Optional< any >`: The entity record's non transient edits.

[getEntityRecords](#)

Returns the Entity's records.

Parameters

- `state State`: State tree
- `kind string`: Entity kind.
- `name string`: Entity name.
- `query GetRecordsHttpQuery`: Optional terms query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for "List [Entity kind]s".

Returns

- `EntityRecord[] | null`: Records.

[getEntityRecordsTotalItems](#)

Returns the Entity's total available records for a given query (ignoring pagination).

Parameters

- `state State`: State tree
- `kind string`: Entity kind.
- `name string`: Entity name.
- `query GetRecordsHttpQuery`: Optional terms query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for "List [Entity kind]s".

Returns

- `number | null`: number | null.

[getEntityRecordsTotalPages](#)

Returns the number of available pages for the given query.

Parameters

- `state State`: State tree
- `kind string`: Entity kind.
- `name string`: Entity name.
- `query GetRecordsHttpQuery`: Optional terms query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for “List [Entity kind]s”.

Returns

- `number | null`: number | null.

[getLastEntityDeleteError](#)

Returns the specified entity record’s last delete error.

Parameters

- `state State`: State tree.
- `kind string`: Entity kind.
- `name string`: Entity name.
- `recordId EntityRecordKey`: Record ID.

Returns

- `any`: The entity record’s save error.

[getLastEntitySaveError](#)

Returns the specified entity record’s last save error.

Parameters

- `state State`: State tree.
- `kind string`: Entity kind.
- `name string`: Entity name.
- `recordId EntityRecordKey`: Record ID.

Returns

- `any`: The entity record’s save error.

[getRawEntityRecord](#)

Returns the entity's record object by key, with its attributes mapped to their raw values.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *key EntityRecordKey*: Record's key.

Returns

- *EntityRecord | undefined*: Object with the entity's raw attributes.

[getRedoEdit](#)

Deprecated since 6.3

Returns the next edit from the current undo offset for the entity records edits history, if any.

Parameters

- *state State*: State tree.

Returns

- *Optional< any >*: The edit.

[getReferenceByDistinctEdits](#)

Returns a new reference when edited values have changed. This is useful in inferring where an edit has been made between states by comparison of the return values using strict equality.

Usage

```
const hasEditOccurred = (
  getReferenceByDistinctEdits( beforeState ) !==
  getReferenceByDistinctEdits( afterState )
);
```

Parameters

- *state Editor state*.

Returns

- A value whose reference will change only when an edit occurs.

[getRevision](#)

Returns a single, specific revision of a parent entity.

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordKey EntityRecordKey*: The key of the entity record whose revisions you want to fetch.
- *revisionKey EntityRecordKey*: The revision's key.
- *query GetRecordsHttpQuery*: Optional query. If requesting specific fields, fields must always include the ID. For valid query parameters see revisions schema in [the REST API Handbook](#). Then see the arguments available “Retrieve a [entity kind]”.

Returns

- RevisionRecord | Record< PropertyKey , never > | undefined: Record.

[**getRevisions**](#)

Returns an entity's revisions.

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordKey EntityRecordKey*: The key of the entity record whose revisions you want to fetch.
- *query GetRecordsHttpQuery*: Optional query. If requesting specific fields, fields must always include the ID. For valid query parameters see revisions schema in [the REST API Handbook](#). Then see the arguments available “Retrieve a [Entity kind]”.

Returns

- RevisionRecord[] | null: Record.

[**getThemeSupports**](#)

Return theme supports data in the index.

Parameters

- *state State*: Data state.

Returns

- any: Index data.

[**getUndoEdit**](#)

Deprecated since 6.3

Returns the previous edit from the current undo offset for the entity records edits history, if any.

Parameters

- *state State*: State tree.

Returns

- `Optional< any >`: The edit.

[getUserPatternCategories](#)

Retrieve the registered user pattern categories.

Parameters

- *state State*: Data state.

Returns

- `Array< UserPatternCategory >`: User patterns category array.

[getUserQueryResults](#)

Returns all the users returned by a query ID.

Parameters

- *state State*: Data state.
- *queryID string*: Query ID.

Returns

- `undefined< 'edit' >[]`: Users list.

[hasEditsForEntityRecord](#)

Returns true if the specified entity record has edits, and false otherwise.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- `boolean`: Whether the entity record has edits or not.

[hasEntityRecords](#)

Returns true if records have been received for the given set of parameters, or false otherwise.

Parameters

- *state State*: State tree

- *kind* string: Entity kind.
- *name* string: Entity name.
- *query GetRecordsHttpQuery*: Optional terms query. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for “List [Entity kind]s”.

Returns

- boolean: Whether entity records have been received.

[hasFetchedAutosaves](#)

Returns true if the REST request for autosaves has completed.

Parameters

- *state State*: State tree.
- *postType* string: The type of the parent post.
- *postId EntityRecordKey*: The id of the parent post.

Returns

- boolean: True if the REST request was completed. False otherwise.

[hasRedo](#)

Returns true if there is a next edit from the current undo offset for the entity records edits history, and false otherwise.

Parameters

- *state State*: State tree.

Returns

- boolean: Whether there is a next edit or not.

[hasUndo](#)

Returns true if there is a previous edit from the current undo offset for the entity records edits history, and false otherwise.

Parameters

- *state State*: State tree.

Returns

- boolean: Whether there is a previous edit or not.

[isAutosavingEntityRecord](#)

Returns true if the specified entity record is autosaving, and false otherwise.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- `boolean`: Whether the entity record is autosaving or not.

[isDeletingEntityRecord](#)

Returns true if the specified entity record is deleting, and false otherwise.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- `boolean`: Whether the entity record is deleting or not.

[isPreviewEmbedFallback](#)

Determines if the returned preview is an oEmbed link fallback.

WordPress can be configured to return a simple link to a URL if it is not embeddable. We need to be able to determine if a URL is embeddable or not, based on what we get back from the oEmbed preview API.

Parameters

- *state State*: Data state.
- *url string*: Embedded URL.

Returns

- `boolean`: Is the preview for the URL an oEmbed link fallback.

[isRequestingEmbedPreview](#)

Returns true if a request is in progress for embed preview data, or false otherwise.

Parameters

- *state State*: Data state.
- *url string*: URL the preview would be for.

Returns

- `boolean`: Whether a request is in progress for an embed preview.

[isSavingEntityRecord](#)

Returns true if the specified entity record is saving, and false otherwise.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- *boolean*: Whether the entity record is saving or not.

[Hooks](#)

The following set of react hooks available to import from the `@wordpress/core-data` package:

[useEntityRecord](#)

Resolves the specified entity record.

Usage

```
import { useEntityRecord } from '@wordpress/core-data';

function PageTitleDisplay( { id } ) {
    const { record, isResolving } = useEntityRecord( 'postType', 'page', id );

    if ( isResolving ) {
        return 'Loading...';
    }

    return record.title;
}

// Rendered in the application:
// <PageTitleDisplay id={ 1 } />
```

In the above example, when `PageTitleDisplay` is rendered into an application, the page and the resolution details will be retrieved from the store state using `getEntityRecord()`, or resolved if missing.

```
import { useCallback } from 'react';
import { useDispatch } from '@wordpress/data';
import { __ } from '@wordpress/i18n';
import { TextControl } from '@wordpress/components';
import { store as noticeStore } from '@wordpress/notices';
import { useEntityRecord } from '@wordpress/core-data';

function PageRenameForm( { id } ) {
```

```

const page = useEntityRecord( 'postType', 'page', id );
const { createSuccessNotice, createErrorNotice } =
  useDispatch( noticeStore );

const setTitle = useCallback(
  ( title ) => {
    page.edit( { title } );
  },
  [ page.edit ]
);

if ( page.isResolving ) {
  return 'Loading...';
}

async function onRename( event ) {
  event.preventDefault();
  try {
    await page.save();
    createSuccessNotice( __( 'Page renamed.' ), {
      type: 'snackbar',
    });
  } catch ( error ) {
    createErrorNotice( error.message, { type: 'snackbar' } );
  }
}

return (
  <form onSubmit={ onRename }>
    <TextControl
      label={ __( 'Name' ) }
      value={ page.editedRecord.title }
      onChange={ setTitle }
    />
    <button type="submit">{ __( 'Save' ) }</button>
  </form>
);
}

// Rendered in the application:
// <PageRenameForm id={ 1 } />

```

In the above example, updating and saving the page title is handled via the `edit()` and `save()` mutation helpers provided by `useEntityRecord()`;

Parameters

- `kind` `string`: Kind of the entity, e.g. `root` or a `postType`. See `rootEntitiesConfig` in `./entities.ts` for a list of available kinds.
- `name` `string`: Name of the entity, e.g. `plugin` or a `post`. See `rootEntitiesConfig` in `./entities.ts` for a list of available names.
- `recordId` `string` | `number`: ID of the requested entity record.
- `options` `Options`: Optional hook options.

Returns

- EntityRecordResolution< RecordType >: Entity record data.

Changelog

6.1.0 Introduced in WordPress core.

useEntityRecords

Resolves the specified entity records.

Usage

```
import { useEntityRecords } from '@wordpress/core-data';

function PageTitlesList() {
    const { records, isResolving } = useEntityRecords( 'postType', 'page' );

    if ( isResolving ) {
        return 'Loading...';
    }

    return (
        <ul>
            { records.map( ( page ) => (
                <li>{ page.title }</li>
            ) ) }
        </ul>
    );
}

// Rendered in the application:
// <PageTitlesList />
```

In the above example, when `PageTitlesList` is rendered into an application, the list of records and the resolution details will be retrieved from the store state using `getEntityRecords()`, or resolved if missing.

Parameters

- `kind` string: Kind of the entity, e.g. `root` or a `postType`. See `rootEntitiesConfig` in `./entities.ts` for a list of available kinds.
- `name` string: Name of the entity, e.g. `plugin` or a `post`. See `rootEntitiesConfig` in `./entities.ts` for a list of available names.
- `queryArgs` Record< string, unknown >: Optional HTTP query description for how to fetch the data, passed to the requested API endpoint.
- `options` Options: Optional hook options.

Returns

- EntityRecordsResolution< RecordType >: Entity records data.

Changelog

6.1.0 Introduced in WordPress core.

[useResourcePermissions](#)

Resolves resource permissions.

Usage

```
import { useResourcePermissions } from '@wordpress/core-data';

function PagesList() {
    const { canCreate, isResolving } = useResourcePermissions( 'pages' );

    if ( isResolving ) {
        return 'Loading ...';
    }

    return (
        <div>
            { canCreate ? <button>+ Create a new page</button> : false }
            // ...
        </div>
    );
}

// Rendered in the application:
// <PagesList />

import { useResourcePermissions } from '@wordpress/core-data';

function Page( { pageId } ) {
    const { canCreate, canUpdate, canDelete, isResolving } =
        useResourcePermissions( 'pages', pageId );

    if ( isResolving ) {
        return 'Loading ...';
    }

    return (
        <div>
            { canCreate ? <button>+ Create a new page</button> : false }
            { canUpdate ? <button>Edit page</button> : false }
            { canDelete ? <button>Delete page</button> : false }
            // ...
        </div>
    );
}

// Rendered in the application:
// <Page pageId={ 15 } />
```

In the above example, when `PagesList` is rendered into an application, the appropriate permissions and the resolution details will be retrieved from the store state using `canUser()`, or resolved if missing.

Parameters

- *resource* `string`: The resource in question, e.g. media.
- *id* `IdType`: ID of a specific resource entry, if needed, e.g. 10.

Returns

- `ResourcePermissionsResolution< IdType >`: Entity records data.

Changelog

6.1.0 Introduced in WordPress core.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/core-data](#)

[Previous](#) [@wordpress/core-commands](#) [Previous: @wordpress/core-commands](#)

[Next](#) [@wordpress/create-block-interactive-template](#) [Next: @wordpress/create-block-interactive-template](#)

@wordpress/create-block-interactive-template

In this article

Table of Contents

- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This is a template for [@wordpress/create-block](#) to create interactive blocks

Usage

This block template can be used by running the following command:

```
npx @wordpress/create-block --template @wordpress/create-block-interactive
```

It requires Gutenberg 17.5 or higher.

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](#).

First published

July 14, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/create-block-interactive-template](#)

[Previous @wordpress/core-data](#) [Previous: @wordpress/core-data](#)

[Next @wordpress/create-block-tutorial-template](#) [Next: @wordpress/create-block-tutorial-template](#)

@wordpress/create-block-tutorial-template

In this article

Table of Contents

- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This is a template for [@wordpress/create-block](#) that creates an example “Copyright Date” block. This block is used in the official WordPress block development [Quick Start Guide](#).

Usage

This block template can be used by running the following command:

```
npx @wordpress/create-block --template @wordpress/create-block-tutorial-template
```

Use the default options when prompted in the terminal.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/create-block-tutorial-template](#)

[Previous @wordpress/create-block-interactive-template](#) Previous: [@wordpress/create-block-interactive-template](#)

Next [@wordpress/create-block](#) Next: [@wordpress/create-block](#)

@wordpress/create-block

In this article

Table of Contents

- [Quick start](#)
- [Usage](#)
 - [Interactive Mode](#)
 - [slug](#)
 - [options](#)
- [Available commands in the scaffolded project](#)
- [External Project Templates](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Create Block is an **officially supported tool for scaffolding a WordPress plugin that registers a block**. It generates PHP, JS, CSS code, and everything you need to start the project. It also integrates a modern build setup with no configuration.

It is largely inspired by [create-react-app](#). Major kudos to [@gaearon](#), the whole Facebook team, and the React community.

Blocks are the fundamental elements of modern WordPress sites. Introduced in [WordPress 5.0](#), they allow [page and post builder-like functionality](#) to every up-to-date WordPress website.

Learn more about the [Block API at the Gutenberg HandBook](#).

Quick start

```
$ npx @wordpress/create-block@latest todo-list  
$ cd todo-list  
$ npm start
```

The `slug` provided (`todo-list` in the example) defines the folder name for the scaffolded plugin and the internal block name. The WordPress plugin generated must [be installed manually](#).

(requires node version 14.0.0 or above, and npm version 6.14.4 or above)

[Watch a video introduction to create-block on Learn.wordpress.org](#)

Usage

The `create-block` command generates a project with PHP, JS, and CSS code for registering a block with a WordPress plugin.

```
$ npx @wordpress/create-block@latest [options] [slug]
```

```
$ np
```

The name for a block is a unique string that identifies a block. Block Names are structured as `namespace/slug`, where namespace is the name of your plugin or theme.

In most cases, we recommended pairing blocks with WordPress plugins rather than themes, because only using plugin ensures that all blocks still work when your theme changes.

[Interactive Mode](#)

When no `slug` is provided, the script will run in interactive mode and will start prompting for the input required (`slug`, `title`, `namespace`...) to scaffold the project.

[slug](#)

The use of `slug` is optional.

When provided it triggers the *quick mode*, where this `slug` is used:

- as the block slug (required for its identification)
- as the output location (folder name) for scaffolded files
- as the name of the WordPress plugin.

The rest of the configuration is set to all default values unless overridden with some options listed below.

options

-V, --version	output the version number
-t, --template <name>	project template type name; allowed values: "scaffold" block files only
--no-plugin	internal namespace for the block name
--namespace <value>	display title for the block and the WordPress
--title <value>	short description for the block and the WordPress
--short-description <value>	category name for the block
--category <name>	enable integration with `@wordpress/scripts`
--wp-scripts	disable integration with `@wordpress/scripts`
--no-wp-scripts	enable integration with `@wordpress/env` package
--wp-env	output usage information
-h, --help	choose a block variant as defined by the temporary environment variable
--variant	
--template	

This argument specifies an *external npm package* as a template.

```
$ npx @wordpress/create-block@latest --template my-template-package
```

This argument also allows to pick a *local directory* as a template.

```
$ npx @wordpress/create-block@latest --template ./path/to/template-directory
```

--variant

With this argument, `create-block` will generate a [dynamic block](#) based on the built-in template.

```
$ npx @wordpress/create-block@latest --variant dynamic  
--help
```

With this argument, the `create-block` package outputs usage information.

```
$ npx @wordpress/create-block@latest --help  
--no-plugin
```

With this argument, the `create-block` package runs in *No plugin mode* which only scaffolds block files into the current directory.

```
$ npx @wordpress/create-block@latest --no-plugin  
--wp-env
```

With this argument, the `create-block` package will add to the generated plugin the configuration and the script to run [wp-env package](#) within the plugin. This will allow you to

easily set up a local WordPress environment (via Docker) for building and testing the generated plugin.

```
$ npx @wordpress/create-block@latest --wp-env
```

[Available commands in the scaffolded project](#)

The plugin folder created when executing this command, is a node package with a modern build setup that requires no configuration.

A set of scripts is available from inside that folder (provided by the `scripts` package) to make your work easier. [Click here](#) for a full description of these commands.

Note: You don't need to install or configure tools like [webpack](#), [Babel](#) or [ESLint](#) yourself. They are preconfigured and hidden so that you can focus on coding.

For example, running the `start` script from inside the generated folder (`npm start`) would automatically start the build for development.

[External Project Templates](#)

[Click here](#) for information on External Project Templates

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: `@wordpress/create-block`](#)

[Previous: `@wordpress/create-block-tutorial-template`](#) [Previous: `@wordpress/create-block-tutorial-template`](#)

[Next: External Project Templates](#) [Next: External Project Templates](#)

External Project Templates

In this article

Table of Contents

- [Project Template Configuration](#)
 - [pluginTemplatesPath](#)
 - [blockTemplatesPath](#)
 - [assetsPath](#)
 - [defaultValues](#)

[↑ Back to top](#)

Are you looking for a way to share your project configuration? Creating an external project template hosted on npm or located in a local directory is possible. These npm packages can provide custom `.mustache` files that replace default files included in the tool for the WordPress plugin or/and the block. It's also possible to override default configuration values used during the scaffolding process.

[Project Template Configuration](#)

Providing the main file (`index.js` by default) for the package that returns a configuration object is mandatory. Several options allow customizing the scaffolding process.

[pluginTemplatesPath](#)

This optional field allows overriding file templates related to **the WordPress plugin shell**. The path points to a location with template files ending with the `.mustache` extension (nested folders are also supported). When not set, the tool uses its own set of templates.

Example:

```
const { join } = require( 'path' );

module.exports = {
    pluginTemplatesPath: join( __dirname, 'plugin-templates' ),
};
```

[blockTemplatesPath](#)

This optional field allows overriding file templates related to **the individual block**. The path points to a location with template files ending with the `.mustache` extension (nested folders are also supported). When not set, the tool uses its own set of templates.

Example:

```
const { join } = require( 'path' );

module.exports = {
```

```
    blockTemplatesPath: join( __dirname, 'block-templates' ) ,  
};
```

assetsPath

This setting is useful when your template scaffolds a WordPress plugin that uses static assets like images or fonts, which should not be processed. It provides the path pointing to the location where assets are located. They will be copied to the `assets` subfolder in the generated plugin.

Example:

```
const { join } = require( 'path' );  
  
module.exports = {  
    assetsPath: join( __dirname, 'plugin-assets' ),  
};
```

defaultValues

It is possible to override the default template configuration using the `defaultValues` field.

Example:

```
module.exports = {  
    defaultValues: {  
        slug: 'my-fantastic-block' ,  
        title: 'My fantastic block' ,  
        dashicon: 'palmtree' ,  
        version: '1.2.3' ,  
    } ,  
};
```

The following configurable variables are used with the template files. Template authors can change default values to use when users don't provide their data.

Project:

- `wpScripts` (default: `true`) – enables integration with the `@wordpress/scripts` package and adds common scripts to the `package.json`.
- `wpEnv` (default: `false`) – enables integration with the `@wordpress/env` package and adds the `env` script to the `package.json`.
- `customScripts` (default: `{}`) – the list of custom scripts to add to `package.json`. It also allows overriding default scripts.
- `npmDependencies` (default: `[]`) – the list of remote npm packages to be installed in the project with [`npm install`](#) when `wpScripts` is enabled.
- `npmDevDependencies` (default: `[]`) – the list of remote npm packages to be installed in the project with [`npm install --save-dev`](#) when `wpScripts` is enabled.
- `customPackageJSON` (no default) – allows definition of additional properties for the generated `package.json` file.

Plugin header fields ([learn more](#)):

- `pluginURI` (no default) – the home page of the plugin.
- `version` (default: `'0.1.0'`) – the current version number of the plugin.

- `author` (default: 'The WordPress Contributors') – the name of the plugin author(s).
- `license` (default: 'GPL-2.0-or-later') – the short name of the plugin's license.
- `licenseURI` (default: '<https://www.gnu.org/licenses/gpl-2.0.html>') – a link to the full text of the license.
- `domainPath` (no default) – a custom domain path for the translations ([more info](#)).
- `updateURI`: (no default) – a custom update URI for the plugin ([related dev note](#)).

Block metadata ([learn more](#)):

- `folderName` (default: '.') – the location for the `block.json` file and other optional block files generated from block templates included in the folder set with the `blockTemplatesPath` setting.
- `$schema` (default: '<https://schemas.wp.org/trunk/block.json>') – the schema URL used for block validation.
- `apiVersion` (default: 2) – the block API version ([related dev note](#)).
- `slug` (no default) – the block slug used for identification in the block name.
- `namespace` (default: 'create-block') – the internal namespace for the block name.
- `title` (no default) – a display title for your block.
- `description` (no default) – a short description for your block.
- `dashicon` (no default) – an icon property that makes it easier to identify a block ([available values](#)).
- `category` (default: 'widgets') – blocks are grouped into categories to help users browse and discover them. The categories provided by core are `text`, `media`, `design`, `widgets`, `theme`, and `embed`.
- `attributes` (no default) – block attributes ([more details](#)).
- `supports` (no default) – optional block extended support features ([more details](#)).
- `editorScript` (default: 'file:./index.js') – an editor script definition.
- `editorStyle` (default: 'file:./index.css') – an editor style definition.
- `style` (default: 'file:./style-index.css') – a frontend and editor style definition.
- `render` (no default) – a path to the PHP file used when rendering the block type on the server before presenting on the front end.
- `customBlockJSON` (no default) – allows definition of additional properties for the generated `block.json` file.

First published

August 31, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: External Project Templates”](#)

[Previous](#) [@wordpress/create-block](#) [Previous: @wordpress/create-block](#)

[Next](#) [@wordpress/custom-templated-path-webpack-plugin](#) [Next: @wordpress/custom-templated-path-webpack-plugin](#)

@wordpress/custom-templated-path-webpack-plugin

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

DEPRECATED for webpack v5: please use [`output.filename`](#) instead.

Webpack plugin for creating custom path template tags. Extend the [default set of template tags](#) with your own custom behavior. Hooks into Webpack's compilation process to allow you to replace tags with a substitute value.

[Installation](#)

Install the module

```
npm install @wordpress/custom-templated-path-webpack-plugin --save-dev
```

Note: This package requires Node.js 12.0.0 or later. It is not compatible with older versions. It works only with webpack v4.

[Usage](#)

Construct an instance of `CustomTemplatedPathPlugin` in your Webpack configurations `plugins` entry, passing an object where keys correspond to the template tag name. The value for each key is a function passed the original intended path and data corresponding to the asset.

The following example creates a new `basename` tag to substitute the basename of each entry file in the build output file. When compiled, the built file will be output as `build-entry.js`.

```
const { basename } = require( 'path' );
const CustomTemplatedPathPlugin = require( '@wordpress/custom-templated-path-webpack-plugin' );

module.exports = {
    // ...
    entry: './entry',
    output: {
        filename: 'build-[basename].js',
    },
}
```

```

plugins: [
    new CustomTemplatedPathPlugin( {
        basename( path, data ) {
            let rawRequest;

            const entryModule = get( data, [ 'chunk', 'entryModule' ] );
            switch ( entryModule.type ) {
                case 'javascript/auto':
                    rawRequest = entryModule.rawRequest;
                    break;

                case 'javascript/esm':
                    rawRequest = entryModule.rootModule.rawRequest;
                    break;
            }

            if ( rawRequest ) {
                return basename( rawRequest );
            }

            return path;
        },
    } ),
],
};

```

For more examples, refer to Webpack's own [TemplatedPathPlugin.js](#), which implements the base set of template tags.

[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](#).

First published

March 9, 2021

Last updated

March 6, 2023

Edit article

[Improve it on GitHub: @wordpress/custom-templated-path-webpack-plugin”](#)

[Previous External Project Templates](#) [Previous: External Project Templates](#)
[Next @wordpress/edit-navigation](#) [Next: @wordpress/edit-navigation](#)

@wordpress/edit-navigation

In this article

Table of Contents

- [Usage](#)
- [Purpose](#)
- [Modes](#)
 - [Classic Mode](#)
 - [Block-based Mode](#)
- [Backwards compatibility](#)
 - [Downgrading from block-based to classic Themes](#)
- [Block to Menu Item mapping](#)
 - [Inconsistencies](#)
- [Hooks](#)
- [Glossary](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Edit Navigation page module for WordPress – a Gutenberg-based UI for editing navigation menus.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

[Usage](#)

```
/**  
 * WordPress dependencies  
 */  
import { initialize } from '@wordpress/edit-navigation';  
  
/**  
 * Internal dependencies  
 */  
import blockEditorSettings from './block-editor-settings';  
  
initialize( '#navigation-editor-root', blockEditorSettings );
```

[Purpose](#)

By default, the Navigation Editor screen allows users to create and edit complex navigations using a block-based UI. The aim is to supersede [the current Menus screen](#) by providing a superior experience whilst retaining backwards compatibility.

The editing experience is provided as a block editor wrapper around the core functionality of the **Navigation block**. Features of the block are disabled/enhanced as necessary to provide an experience appropriate to editing a navigation outside of a Full Site Editing context.

Modes

The Navigation Editor has two “modes” for *persistence* (“saving” navigations) and *rendering*:

1. **Classic (default)** – navigations are saved to the *existing* (post type powered) Menus system and rendered using standard [Walker](#) classes.
2. **Block-based** (opt *in*) – navigations continue to be *saved* using the existing post type system, but:
 - the [navigation is rendered using the core/navigation block](#) (as opposed to [Walker](#)) to provide access to the full power of blocks (with some tradeoffs in terms of backwards compatibility).
 - non-link blocks (anything that is not `core/navigation-link`) are saved as *blocks*.

Classic Mode

In this mode, navigations created in the Navigation Editor are stored using the *existing Menu post type* (`nav_menu_item`) system. As this method matches that used in the *existing* Menus screen, there is a smooth upgrade path to using new Navigation Editor screen to edit navigations.

Moreover, when the navigation is rendered on the front of the site the system continues to use [the classic Navigation “Walker” class](#), thereby ensuring the HTML markup remains the same when using a classic Theme.

Block-based Mode

Important: block-based mode has been temporarily *disabled* until it becomes stable. So, if a theme declares support for the `block-nav-menus` feature it will not affect the frontend.

If desired, themes are able to opt into [rendering complete block-based menus](#) using the Navigation Editor. This allows for arbitrarily complex navigation block structures to be used in an existing theme whilst still ensuring the navigation data is still *saved* to the existing (post type powered) Menus system.

Themes can opt into this behaviour by declaring:

```
add_theme_support( 'block-nav-menus' );
```

This unlocks significant additional capabilities in the Navigation Editor. For example, by default, [the Navigation Editor screen only allows link \(core/navigation-link\) blocks to be inserted into a navigation](#). When a theme opts into `block-nav-menus` however, users are able to add non-link blocks to a navigation using the Navigation Editor screen, including:

- `core/navigation-link`.
- `core/social`.
- `core/search`.

As these items are still saved to `nav_menu_items` this ensures if we ever revert to classic ([Walker](#)-based) rendering, these items will still be rendered (as blocks).

Backwards compatibility

By design the underlying systems of the Nav Editor screen should be largely backwards compatible with the existing Menus screen. Therefore any navigations created or edited using the new Navigation Editor screen should continue to work in the existing classic Menus screen.

Currently, the only exception to this would be any custom functionality added (by Plugins or otherwise) to the existing Menus screen would not be replicated in the new Navigation Editor screen. In this scenario there might be danger of some data loss.

Downgrading from block-based to classic Themes

If the user switches to a theme that does not support block menus, or disables this functionality, ~non-link blocks are no longer rendered on the frontend~ [block-based links will still be rendered on the front end](#). Care is also taken to ensure that users can still see their data on the existing Menus screen.

Block to Menu Item mapping

The Navigation Editor needs to be able to map navigation items in two directions:

1. `nav_menu_items` to Blocks – when displaying an existing navigation.
2. Blocks to `nav_menu_items` – when *saving* an navigation being editing in the Navigation screen.

The Navigation Editor has two dedicated methods for handling mapping between these two expressions of the data:

- [`menuItemToBlockAttributes\(\)`](#).
- [`blockAttributestoMenuItem\(\)`](#)

To understand these fully, one must appreciate that WordPress maps raw `nav_menu_item` posts to [Menu item objects](#). These have various properties which map as follows:

Menu Item object property	Equivalent Block Attribute	Description
ID	Not mapped.	The term_id if the menu item represents a taxonomy term.
attr_title	title	The title attribute of the link element for this menu item.
classes	classNames	The array of class attribute values for the link element of this menu item.
db_id	Not mapped.	The DB ID of this item as a nav_menu_item object, if it exists (0 if it doesn't exist).
description	description	The description of this menu item.
menu_item_parent	Not mapped. ¹	The DB ID of the nav_menu_item that is this item's menu parent, if any. 0 otherwise.
object	type	

Menu Item object property	Equivalent Block Attribute	Description
object_id	id	The type of object originally represented, such as ‘category’, ‘post’, or ‘attachment’.
post_parent	Not mapped.	The DB ID of the original object this menu item represents, e.g. ID for posts and term_id for categories.
post_title	Not mapped.	The DB ID of the original object’s parent object, if any (0 otherwise).
target	opensInNewTab ²	A “no title” label if menu item represents a post that lacks a title.
title	label	The target attribute of the link element for this menu item.
type	kind	The title of this menu item.
type_label	Not mapped.	The family of objects originally represented, such as ‘post_type’ or ‘taxonomy’.
url	url	The singular label used to describe this type of menu item.
xfn	rel	The URL to which this menu item points.
_invalid	Not mapped.	The XFN relationship expressed in the link of this menu item.
		Whether the menu item represents an object that no longer exists.

- [1] – the parent -> child relationship is expressed in block via the innerBlocks attribute and is therefore not required as a explicit block attribute.
- [2] – applies only if the value of the target field is _blank.

Inconsistencies

Mapping

For historical reasons, the following properties display some inconsistency in their mapping from Menu Item Object to Block attribute:

- type -> kind – the family of objects is stored as kind on the block and so must be mapped accordingly.
- object -> type – the type of object is stored as type on the block and so must be mapped accordingly.
- object_id -> id – the block stores a reference to the original object’s ID as the id attribute. This should not be confused with the block’s clientId which is unrelated.
- attr_title -> title – the HTML title attribute is stored as title on the block and so must be mapped accordingly.

Object Types

- Menu Item objects which represent “Tags” are stored in WordPress as `post_tag` but the block expects their `type` attribute to be `tag` (omiting the `post_` suffix). This inconsistency is accounted for in [the mapping utilities methods](#).

Hooks

The `useNavigationEditor` and `useEntityBlockEditor` hooks are the central part of this package. They bridge the gap between the API and the block editor interface:

```
// Data from API:
const {
  menus,
  hasLoadedMenus,
  selectedMenuItemId,
  navigationPost,
} = useNavigationEditor();

// Working state:
const [ blocks, onInput, onChange ] = useEntityBlockEditor(
  NAVIGATION_POST_KIND,
  NAVIGATION_POST_POST_TYPE,
  {
    id: navigationPost?.id,
  }
);

const isBlockEditorReady = !! (
  menus?.length &&
  navigationPost &&
  selectedMenuItemId
);

return (
  <BlockEditorProvider
    value={ blocks }
    onInput={ onInput }
    onChange={ onChange }
    settings={ blockEditorSettings }
  >
  { isBlockEditorReady && (
    <div className="edit-navigation-layout__content-area">
      <BlockTools>
        <Editor isPending={ ! hasLoadedMenus } />
      </BlockTools>
    </div>
  ) }
</BlockEditorProvider>
);
```

Glossary

- **(Navigation) link** – the basic `core/navigation-link` block which is the standard block used to add links within navigations.
- **Block-based link** – any navigation item that is *not* a `core/navigation-link` block. These are persisted as blocks but still utilise the existing Menus post type system.
- **Navigation block** – the root `core/navigation` block which can be used both with the Navigation Editor and outside (eg: Post / Site Editor).
- **Navigation editor / screen** – the new screen provided by Gutenberg to allow the user to edit navigations using a block-based UI.
- **Menus screen** – the current/existing [interface/screen for managing Menus](#) in WordPress WPAdmin.

*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.*

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](#).

First published

March 9, 2021

Last updated

January 26, 2023

Edit article

[Improve it on GitHub: @wordpress/edit-navigation](#)

Previous: [@wordpress/custom-templated-path-webpack-plugin](#) Previous: [@wordpress/custom-templated-path-webpack-plugin](#)

Next: [@wordpress/customize-widgets](#) Next: [@wordpress/customize-widgets](#)

@wordpress/customize-widgets

In this article

Table of Contents

- [Installation](#)
- [Technical implementation details](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Widgets blocks in Customizer Module for WordPress.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

Installation

Install the module

```
npm install @wordpress/customize-widgets
```

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.

Technical implementation details

The new Widgets Customizer replaces Appearance > Customize > Widgets with block-based editors. The original Customizer is a Backbone app, but the new editor is a React app. One of the challenges is to integrate them together but make sure features from both sides still work.

We extend the Customizer's sections and controls in the `/controls` directory and inject some custom logic for the editor. We use React portal to render each editor in its section to reuse most of the styles and scripts provided by the Customizer.

`components/sidebar-block-editor` is the entry point for each widget area's block editor. `component/sidebar-block-editor/sidebar-adapter.js` is an adapter to talk to [the Customize API](#) and transform widget objects into widget instances.

`components/sidebar-block-editor/use-sidebar-block-editor.js` is a custom React Hook to integrate the adapter into React and handle most of the translations between blocks and widgets. These allow us to implement basic editing features as well as real-time preview in a backwards-compatible way.

Whenever the blocks change, we run through each block to determine if there are created, edited, or deleted blocks. We then convert them to their widget counterparts and call the Customize API to update them.

For React developers, this can be thought of as a custom reconciler or a custom renderer for the Customizer. But instead of targeting DOM as the render target, we are targeting WordPress widgets using the Customize API.

This is not the typical way the block editor is intended to be used. As a result, we have to also implement some missing features such as undo/redo and custom focus control. It is still a goal to make the block editor as easy to integrate into different systems as possible, so the integration in the Widgets Customizer can be a good experience for us to reflect some drawbacks in our current API and potentially improve them in the future.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/customize-widgets”](#)

[Previous @wordpress/edit-navigation](#) Previous: [@wordpress/edit-navigation](#)
[Next @wordpress/data-controls](#) Next: [@wordpress/data-controls](#)

@wordpress/data-controls

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [apiFetch](#)
 - [controls](#)
 - [dispatch](#)
 - [select](#)
 - [syncSelect](#)
- [Contributing to this package](#)

[↑ Back to top](#)

The data controls module is a module intended to simplify implementation of common controls used with the [@wordpress/data](#) package.

Note: It is assumed that the registry being used has the controls plugin enabled on it (see [more details on controls here](#))

Installation

Install the module

```
npm install @wordpress/data-controls --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.*

The following controls are available on the object returned by the module:

API

apiFetch

Dispatches a control action for triggering an api fetch call.

Usage

```
import { apiFetch } from '@wordpress/data-controls';

// Action generator using apiFetch
export function* myAction() {
    const path = '/v2/my-api/items';
    const items = yield apiFetch( { path } );
    // do something with the items.
}
```

Parameters

- *request Object*: Arguments for the fetch request.

Returns

- *Object*: The control descriptor.

controls

The default export is what you use to register the controls with your custom store.

Usage

```
// WordPress dependencies
import { controls } from '@wordpress/data-controls';
import { registerStore } from '@wordpress/data';

// Internal dependencies
import reducer from './reducer';
import * as selectors from './selectors';
import * as actions from './actions';
import * as resolvers from './resolvers';
```

```
registerStore( 'my-custom-store', {  
    reducer,  
    controls,  
    actions,  
    selectors,  
    resolvers,  
} );
```

Returns

- **Object**: An object for registering the default controls with the store.

[dispatch](#)

Control for dispatching an action in a registered data store. Alias for the `dispatch` control in the `@wordpress/data` package.

Parameters

- `storeNameOrDescriptor string | StoreDescriptor`: The store object or identifier.
- `actionName string`: The action name.
- `args any []`: Arguments passed without change to the `@wordpress/data` control.

[select](#)

Control for resolving a selector in a registered data store. Alias for the `resolveSelect` built-in control in the `@wordpress/data` package.

Parameters

- `storeNameOrDescriptor string | StoreDescriptor`: The store object or identifier.
- `selectorName string`: The selector name.
- `args any []`: Arguments passed without change to the `@wordpress/data` control.

[syncSelect](#)

Control for calling a selector in a registered data store. Alias for the `select` built-in control in the `@wordpress/data` package.

Parameters

- `storeNameOrDescriptor string | StoreDescriptor`: The store object or identifier.
- `selectorName string`: The selector name.
- `args any []`: Arguments passed without change to the `@wordpress/data` control.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/data-controls”](#)

[Previous @wordpress/customize-widgets](#) [Previous: @wordpress/customize-widgets](#)
[Next @wordpress/data](#) [Next: @wordpress/data](#)

@wordpress/data

In this article

[Table of Contents](#)

- [Installation](#)
- [Registering a Store](#)
 - [Redux Store Options](#)
- [Generic Stores](#)
- [Comparison with Redux](#)
- [API](#)
 - [AsyncModeProvider](#)
 - [combineReducers](#)
 - [controls](#)
 - [createReduxStore](#)
 - [createRegistry](#)
 - [createRegistryControl](#)
 - [createRegistrySelector](#)
 - [dispatch](#)
 - [plugins](#)
 - [register](#)
 - [registerGenericStore](#)
 - [registerStore](#)
 - [RegistryConsumer](#)
 - [RegistryProvider](#)
 - [resolveSelect](#)
 - [select](#)
 - [subscribe](#)
 - [suspendSelect](#)
 - [use](#)
 - [useDispatch](#)
 - [useRegistry](#)
 - [useSelect](#)
 - [useSuspenseSelect](#)

- [withDispatch](#)
- [withRegistry](#)
- [withSelect](#)
- [batch](#)
- [Selectors](#)
 - [hasFinishedResolution](#)
 - [hasStartedResolution](#)
 - [isResolving](#)
 - [Normalizing Selector Arguments](#)
- [Going further](#)
- [Contributing to this package](#)

[↑ Back to top](#)

WordPress' data module serves as a hub to manage application state for both plugins and WordPress itself, providing tools to manage data within and between distinct modules. It is designed as a modular pattern for organizing and sharing data: simple enough to satisfy the needs of a small plugin, while scalable to serve the requirements of a complex single-page application.

The data module is built upon and shares many of the same core principles of [Redux](#), but shouldn't be mistaken as merely *Redux for WordPress*, as it includes a few of its own [distinguishing characteristics](#). As you read through this guide, you may find it useful to reference the Redux documentation — particularly [its glossary](#) — for more detail on core concepts.

[Installation](#)

Install the module

```
npm install @wordpress/data --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.*

[Registering a Store](#)

Use the `register` function to add your own store to the centralized data registry. This function accepts one argument – a store descriptor that can be created with `createReduxStore` factory function. `createReduxStore` accepts two arguments: a name to identify the module, and a configuration object with values describing how your state is represented, modified, and accessed. At a minimum, you must provide a reducer function describing the shape of your state and how it changes in response to actions dispatched to the store.

```
import apiFetch from '@wordpress/api-fetch';
import { createReduxStore, register } from '@wordpress/data';

const DEFAULT_STATE = {
  prices: {},
  discountPercent: 0,
};

const actions = {
```

```

setPrice( item, price ) {
    return {
        type: 'SET_PRICE',
        item,
        price,
    };
},
startSale( discountPercent ) {
    return {
        type: 'START_SALE',
        discountPercent,
    };
},
fetchFromAPI( path ) {
    return {
        type: 'FETCH_FROM_API',
        path,
    };
},
};

const store = createReduxStore( 'my-shop', {
    reducer( state = DEFAULT_STATE, action ) {
        switch ( action.type ) {
            case 'SET_PRICE':
                return {
                    ...state,
                    prices: {
                        ...state.prices,
                        [ action.item ]: action.price,
                    },
                };
            case 'START_SALE':
                return {
                    ...state,
                    discountPercent: action.discountPercent,
                };
        }
        return state;
    },
    actions,
    selectors: {
        getPrice( state, item ) {
            const { prices, discountPercent } = state;
            const price = prices[ item ];

            return price * ( 1 - 0.01 * discountPercent );
        },
    },
}
);

```

```

    } ,

  controls: {
    FETCH_FROM_API( action ) {
      return apiFetch( { path: action.path } );
    },
  },

  resolvers: {
    *getPrice( item ) {
      const path = '/wp/v2/prices/' + item;
      const price = yield actions.fetchFromAPI( path );
      return actions.setPrice( item, price );
    },
  },
} );
}

register( store );

```

The return value of `createReduxStore` is the `StoreDescriptor` object that contains two properties:

- `name (string)` – the name of the store
- `instantiate (Function)` – it returns a [Redux-like store object](#) with the following methods:
 - `getState ()`: Returns the state value of the registered reducer
 - *Redux parallel:* [getState](#)
 - `subscribe(listener: Function)`: Registers a function called any time the value of state changes.
 - *Redux parallel:* [subscribe](#)
 - `dispatch(action: Object)`: Given an action object, calls the registered reducer and updates the state value.
 - *Redux parallel:* [dispatch](#)

[Redux Store Options](#)

`reducer`

A [reducer](#) is a function accepting the previous `state` and `action` as arguments and returns an updated `state` value.

`actions`

The `actions` object should describe all [action creators](#) available for your store. An action creator is a function that optionally accepts arguments and returns an action object to dispatch to the registered reducer. *Dispatching actions is the primary mechanism for making changes to your state.*

`selectors`

The `selectors` object includes a set of functions for accessing and deriving state values. A selector is a function which accepts state and optional arguments and returns some value from state. *Calling selectors is the primary mechanism for retrieving data from your state,* and serve as

a useful abstraction over the raw data which is typically more susceptible to change and less readily usable as a [normalized object](#).

resolvers

A **resolver** is a side-effect for a selector. If your selector result may need to be fulfilled from an external source, you can define a resolver such that the first time the selector is called, the fulfillment behavior is effected.

The `resolvers` option should be passed as an object where each key is the name of the selector to act upon, the value a function which receives the same arguments passed to the selector, excluding the state argument. It can then dispatch as necessary to fulfill the requirements of the selector, taking advantage of the fact that most data consumers will subscribe to subsequent state changes (by `subscribe` or `withSelect`).

controls

A **control** defines the execution flow behavior associated with a specific action type. This can be particularly useful in implementing asynchronous data flows for your store. By defining your action creator or resolvers as a generator which yields specific controlled action types, the execution will proceed as defined by the control handler.

The `controls` option should be passed as an object where each key is the name of the action type to act upon, the value a function which receives the original action object. It should return either a promise which is to resolve when evaluation of the action should continue, or a value. The value or resolved promise value is assigned on the return value of the yield assignment. If the control handler returns undefined, the execution is not continued.

Refer to the [documentation of @wordpress/redux-routine](#) for more information.

initialState

An optional preloaded initial state for the store. You may use this to restore some serialized state value or a state generated server-side.

Generic Stores

The `@wordpress/data` module offers a more advanced and generic interface for the purposes of integrating other data systems and situations where more direct control over a data system is needed. In this case, a data store will need to be implemented outside of `@wordpress/data` and then plugged in via three functions:

- `getSelectors()`: Returns an object of selector functions, pre-mapped to the store.
- `getActions()`: Returns an object of action functions, pre-mapped to the store.
- `subscribe(listener: Function)`: Registers a function called any time the value of state changes.
 - Behaves as Redux [subscribe](#) with the following differences:
 - Doesn't have to implement an unsubscribe, since the registry never uses it.
 - Only has to support one listener (the registry).

By implementing the above interface for your custom store, you gain the benefits of using the registry and the `withSelect` and `withDispatch` higher order components in your application code. This provides seamless integration with existing and alternative data systems.

Integrating an existing redux store with its own reducers, store enhancers and middleware can be accomplished as follows:

Example:

```
import { register } from '@wordpress/data';
import existingSelectors from './existing-app/selectors';
import existingActions from './existing-app/actions';
import createStore from './existing-app/store';

const reduxStore = createStore();

const mapValues = ( obj, callback ) =>
    Object.entries( obj ).reduce(
        ( acc, [ key, value ] ) => ( {
            ...acc,
            [ key ]: callback( value ),
        } ),
        {}
    );
);

const boundSelectors = mapValues(
    existingSelectors,
    ( selector ) =>
        ( ...args ) =>
            selector( reduxStore.getState(), ...args )
);
;

const boundActions = mapValues(
    existingActions,
    ( action ) =>
        ( ...args ) =>
            reduxStore.dispatch( action( ...args ) )
);
;

const genericStore = {
    name: 'existing-app',
    instantiate: () => ( {
        getSelectors: () => boundSelectors,
        getActions: () => boundActions,
        subscribe: reduxStore.subscribe,
    } ),
};
;

register( genericStore );
```

It is also possible to implement a completely custom store from scratch:

Example:

```

import { register } from '@wordpress/data';

function customStore() {
    return {
        name: 'custom-data',
        instantiate: () => {
            const listeners = new Set();
            const prices = { hammer: 7.5 };

            function storeChanged() {
                for ( const listener of listeners ) {
                    listener();
                }
            }

            function subscribe( listener ) {
                listeners.add( listener );
                return () => listeners.delete( listener );
            }

            const selectors = {
                getPrice( itemName ) {
                    return prices[ itemName ];
                },
            };

            const actions = {
                setPrice( itemName, price ) {
                    prices[ itemName ] = price;
                    storeChanged();
                },
            };

            return {
                getSelectors: () => selectors,
                getActions: () => actions,
                subscribe,
            };
        },
    };
}

register( customStore );

```

Comparison with Redux

The data module shares many of the same [core principles](#) and [API method naming](#) of [Redux](#). In fact, it is implemented atop Redux. Where it differs is in establishing a modularization pattern for creating separate but interdependent stores, and in codifying conventions such as selector functions as the primary entry point for data access.

The [higher-order components](#) were created to complement this distinction. The intention with splitting `withSelect` and `withDispatch` — where in React Redux they are combined under

`connect` as `mapStateToProps` and `mapDispatchToProps` arguments — is to more accurately reflect that dispatch is not dependent upon a subscription to state changes, and to allow for state-derived values to be used in `withDispatch` (via [higher-order component composition](#)).

The data module also has built-in solutions for handling asynchronous side-effects, through [resolvers](#) and [controls](#). These differ slightly from [standard redux async solutions](#) like [redux-thunk](#) or [redux-saga](#).

Specific implementation differences from Redux and React Redux:

- In Redux, a `subscribe` listener is called on every dispatch, regardless of whether the value of state has changed.
 - In `@wordpress/data`, a subscriber is only called when state has changed.
- In React Redux, a `mapStateToProps` function must return an object.
 - In `@wordpress/data`, a `withSelect` mapping function can return `undefined` if it has no props to inject.
- In React Redux, the `mapDispatchToProps` argument can be defined as an object or a function.
 - In `@wordpress/data`, the `withDispatch` higher-order component creator must be passed a function.

API

[AsyncModeProvider](#)

Context Provider Component used to switch the data module component rerendering between Sync and Async modes.

Usage

```
import { useSelect, AsyncModeProvider } from '@wordpress/data';
import { store as blockEditorStore } from '@wordpress/block-editor';

function BlockCount() {
    const count = useSelect( ( select ) => {
        return select( blockEditorStore ).getBlockCount();
    }, [] );
    return count;
}

function App() {
    return (
        <AsyncModeProvider value={ true }>
            <BlockCount />
        </AsyncModeProvider>
    );
}
```

In this example, the `BlockCount` component is rerendered asynchronously. It means if a more critical task is being performed (like typing in an input), the rerendering is delayed until the browser becomes IDLE.

It is possible to nest multiple levels of `AsyncModeProvider` to fine-tune the rendering behavior.

Parameters

- `props.value boolean`: Enable Async Mode.

Returns

- `Component`: The component to be rendered.

combineReducers

The combineReducers helper function turns an object whose values are different reducing functions into a single reducing function you can pass to registerReducer.

Usage

```
import { combineReducers, createReduxStore, register } from '@wordpress/data'

const prices = ( state = {}, action ) => {
    return action.type === 'SET_PRICE'
        ? {
            ...state,
            [ action.item ]: action.price,
        }
        : state;
};

const discountPercent = ( state = 0, action ) => {
    return action.type === 'START_SALE' ? action.discountPercent : state;
};

const store = createReduxStore( 'my-shop', {
    reducer: combineReducers( {
        prices,
        discountPercent,
    } ),
} );
register( store );
```

Type

- `import('./types').combineReducers`

Parameters

- `reducers Object`: An object whose values correspond to different reducing functions that need to be combined into one.

Returns

- `Function`: A reducer that invokes every reducer inside the reducers object, and constructs a state object with the same shape.

[controls](#)

Undocumented declaration.

[createReduxStore](#)

Creates a data store descriptor for the provided Redux store configuration containing properties describing reducer, actions, selectors, controls and resolvers.

Usage

```
import { createReduxStore } from '@wordpress/data';

const store = createReduxStore( 'demo', {
    reducer: ( state = 'OK' ) => state,
    selectors: {
        getValue: ( state ) => state,
    },
} );
```

Parameters

- *key string*: Unique namespace identifier.
- *options ReduxStoreConfig<State,Actions,Selectors>*: Registered store options, with properties describing reducer, actions, selectors, and resolvers.

Returns

- *StoreDescriptor<ReduxStoreConfig<State,Actions,Selectors>>*: Store Object.

[createRegistry](#)

Creates a new store registry, given an optional object of initial store configurations.

Parameters

- *storeConfigs Object*: Initial store configurations.
- *parent Object?*: Parent registry.

Returns

- *WPDataRegistry*: Data registry.

[createRegistryControl](#)

Creates a control function that takes additional curried argument with the `registry` object. While a regular control has signature

```
( action ) => iteratorOrPromise;
```

where the control works with the `action` that it's bound to, a registry control has signature:

```
( registry ) => ( action ) => iteratorOrPromise;
```

A registry control is typically used to select data or dispatch an action to a registered store.

When registering a control created with `createRegistryControl` with a store, the store knows which calling convention to use when executing the control.

Parameters

- `registryControl Function`: Function receiving a registry object and returning a control.

Returns

- `Function`: Registry control that can be registered with a store.

[createRegistrySelector](#)

Creates a selector function that takes additional curried argument with the registry `select` function. While a regular selector has signature

```
( state, ...selectorArgs ) => result;
```

that allows to select data from the store's `state`, a registry selector has signature:

```
( select ) =>
  ( state, ...selectorArgs ) =>
    result;
```

that supports also selecting from other registered stores.

Usage

```
import { store as coreStore } from '@wordpress/core-data';
import { store as editorStore } from '@wordpress/editor';

const getCurrentPostId = createRegistrySelector( ( select ) => ( state ) =>
  return select( editorStore ).getCurrentPostId();
} );

const getPostEdits = createRegistrySelector( ( select ) => ( state ) => {
  // calling another registry selector just like any other function
  const postType = getCurrentPostType( state );
  const postId = getCurrentPostId( state );
  return select( coreStore ).getEntityRecordEdits(
    'postType',
    postType,
    postId
  );
} );
```

Note how the `getCurrentPostId` selector can be called just like any other function, (it works even inside a regular non-registry selector) and we don't need to pass the registry as argument. The registry binding happens automatically when registering the selector with a store.

Parameters

- *registrySelector* Function: Function receiving a registry `select` function and returning a state selector.

Returns

- Function: Registry selector that can be registered with a store.

dispatch

Given a store descriptor, returns an object of the store's action creators. Calling an action creator will cause it to be dispatched, updating the state value accordingly.

Note: Action creators returned by the dispatch will return a promise when they are called.

Usage

```
import { dispatch } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

dispatch( myCustomStore ).setPrice( 'hammer', 9.75 );
```

Parameters

- *storeNameOrDescriptor* `StoreNameOrDescriptor`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `DispatchReturn< StoreNameOrDescriptor >`: Object containing the action creators.

plugins

Object of available plugins to use with a registry.

Related

- [use](#)

Type

- `Object`

register

Registers a standard @wordpress/data store descriptor.

Usage

```
import { createReduxStore, register } from '@wordpress/data';

const store = createReduxStore( 'demo', {
    reducer: ( state = 'OK' ) => state,
```

```
        selectors: {
          getValue: ( state ) => state,
        },
      } );
register( store );
```

Parameters

- *store* `StoreDescriptor`: Store descriptor.

[registerGenericStore](#)

Deprecated Use `register(storeDescriptor)` instead.

Registers a generic store instance.

Parameters

- *name* `string`: Store registry name.
- *store* `Object`: Store instance ({ `getSelectors`, `getActions`, `subscribe` }).

[registerStore](#)

Deprecated Use `register` instead.

Registers a standard @wordpress/data store.

Parameters

- *storeName* `string`: Unique namespace identifier for the store.
- *options* `Object`: Store description (reducer, actions, selectors, resolvers).

Returns

- `Object`: Registered store object.

[RegistryConsumer](#)

A custom react Context consumer exposing the provided `registry` to children components.
Used along with the `RegistryProvider`.

You can read more about the react context api here: <https://reactjs.org/docs/context.html#contextprovider>

Usage

```
import {
  RegistryProvider,
  RegistryConsumer,
  createRegistry
} from '@wordpress/data';

const registry = createRegistry( {} );
```

```

const App = ( { props } ) => {
  return <RegistryProvider value={ registry }>
    <div>Hello There</div>
    <RegistryConsumer>
      { ( registry ) => (
        <ComponentUsingRegistry
          { ...props }
          registry={ registry }
        ) }
    </RegistryConsumer>
  </RegistryProvider>
}

```

[RegistryProvider](#)

A custom Context provider for exposing the provided `registry` to children components via a consumer.

See `RegistryConsumer` documentation for example.

[resolveSelect](#)

Given a store descriptor, returns an object containing the store's selectors pre-bound to state so that you only need to supply additional arguments, and modified so that they return promises that resolve to their eventual values, after any resolvers have ran.

Usage

```

import { resolveSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

resolveSelect( myCustomStore ).getPrice( 'hammer' ).then( console.log );

```

Parameters

- `storeNameOrDescriptor StoreDescriptor | string`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `Object`: Object containing the store's promise-wrapped selectors.

[select](#)

Given a store descriptor, 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.

Usage

```

import { select } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

select( myCustomStore ).getPrice( 'hammer' );

```

Parameters

- `storeNameOrDescriptor string | T`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `CurriedSelectorsOf< T >`: Object containing the store's selectors.

[subscribe](#)

Given a listener function, the function will be called any time the state value of one of the registered stores has changed. If you specify the optional `storeNameOrDescriptor` parameter, the listener function will be called only on updates on that one specific registered store.

This function returns an `unsubscribe` function used to stop the subscription.

Usage

```
import { subscribe } from '@wordpress/data';

const unsubscribe = subscribe( () => {
    // You could use this opportunity to test whether the derived result of
    // selector has subsequently changed as the result of a state update.
} );

// Later, if necessary...
unsubscribe();
```

Parameters

- `listener Function`: Callback function.
- `storeNameOrDescriptor string | StoreDescriptor?`: Optional store name.

[suspendSelect](#)

Given a store descriptor, returns an object containing the store's selectors pre-bound to state so that you only need to supply additional arguments, and modified so that they throw promises in case the selector is not resolved yet.

Parameters

- `storeNameOrDescriptor StoreDescriptor | string`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `Object`: Object containing the store's suspense-wrapped selectors.

[use](#)

Extends a registry to inherit functionality provided by a given plugin. A plugin is an object with properties aligning to that of a registry, merged to extend the default registry behavior.

Parameters

- *plugin Object*: Plugin object.

useDispatch

A custom react hook returning the current registry dispatch actions creators.

Note: The component using this hook must be within the context of a RegistryProvider.

Usage

This illustrates a pattern where you may need to retrieve dynamic data from the server via the `useSelect` hook to use in combination with the `dispatch` action.

```
import { useCallback } from 'react';
import { useDispatch, useSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function Button( { onClick, children } ) {
    return (
        <button type="button" onClick={ onClick }>
            { children }
        </button>
    );
}

const SaleButton = ( { children } ) => {
    const { stockNumber } = useSelect(
        ( select ) => select( myCustomStore ).getStockNumber(),
        []
    );
    const { startSale } = useDispatch( myCustomStore );
    const onClick = useCallback( () => {
        const discountPercent = stockNumber > 50 ? 10 : 20;
        startSale( discountPercent );
    }, [ stockNumber ] );
    return <Button onClick={ onClick }>{ children }</Button>;
};

// Rendered somewhere in the application:
// 
// <SaleButton>Start Sale!</SaleButton>
```

Parameters

- *storeNameOrDescriptor [StoreNameOrDescriptor]*: Optionally provide the name of the store or its descriptor from which to retrieve action creators. If not provided, the `registry.dispatch` function is returned instead.

Returns

- `UseDispatchReturn<StoreNameOrDescriptor>`: A custom react hook.

[useRegistry](#)

A custom react hook exposing the registry context for use.

This exposes the `registry` value provided via the [Registry Provider](#) to a component implementing this hook.

It acts similarly to the `useContext` react hook.

Note: Generally speaking, `useRegistry` is a low level hook that in most cases won't be needed for implementation. Most interactions with the `@wordpress/data` API can be performed via the `useSelect` hook, or the `withSelect` and `withDispatch` higher order components.

Usage

```
import { RegistryProvider, createRegistry, useRegistry } from '@wordpress/data';

const registry = createRegistry( {} );

const SomeChildUsingRegistry = ( props ) => {
    const registry = useRegistry();
    // ...logic implementing the registry in other react hooks.
};

const ParentProvidingRegistry = ( props ) => {
    return (
        <RegistryProvider value={ registry }>
            <SomeChildUsingRegistry { ...props } />
        </RegistryProvider>
    );
};
```

Returns

- **Function:** A custom react hook exposing the registry context value.

[useSelect](#)

Custom react hook for retrieving props from registered selectors.

In general, this custom React hook follows the [rules of hooks](#).

Usage

```
import { useSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function HammerPriceDisplay( { currency } ) {
    const price = useSelect(
        ( select ) => {
            return select( myCustomStore ).getPrice( 'hammer', currency );
        },
        [ currency ]
```

```

);
return new Intl.NumberFormat( 'en-US', {
  style: 'currency',
  currency,
} ).format( price );
}

// Rendered in the application:
// <HammerPriceDisplay currency="USD" />

```

In the above example, when `HammerPriceDisplay` is rendered into an application, the price will be retrieved from the store state using the `mapSelect` callback on `useSelect`. If the `currency` prop changes then any price in the state for that currency is retrieved. If the `currency` prop doesn't change and other props are passed in that do change, the price will not change because the dependency is just the `currency`.

When data is only used in an event callback, the data should not be retrieved on render, so it may be useful to get the `selectors` function instead.

Don't use `useSelect` this way when calling the `selectors` in the render function because your component won't re-render on a data change.

```

import { useSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function Paste( { children } ) {
  const { getSettings } = useSelect( myCustomStore );
  function onPaste() {
    // Do something with the settings.
    const settings = getSettings();
  }
  return <div onPaste={ onPaste }>{ children }</div>;
}

```

Parameters

- `mapSelect T`: Function called on every state change. The returned value is exposed to the component implementing this hook. The function receives the `registry.select` method on the first argument and the `registry` on the second argument. When a store key is passed, all selectors for the store will be returned. This is only meant for usage of these selectors in event callbacks, not for data needed to create the element tree.
- `deps unknown[]`: If provided, this memoizes the `mapSelect` so the same `mapSelect` is invoked on every state change unless the dependencies change.

Returns

- `UseSelectReturn<T>`: A custom react hook.

[useSuspenseSelect](#)

A variant of the `useSelect` hook that has the same API, but will throw a suspense Promise if any of the called selectors is in an unresolved state.

Parameters

- *mapSelect* Function: Function called on every state change. The returned value is exposed to the component using this hook. The function receives the `registry.suspendSelect` method as the first argument and the `registry` as the second one.
- *deps* Array: A dependency array used to memoize the `mapSelect` so that the same `mapSelect` is invoked on every state change unless the dependencies change.

Returns

- `Object`: Data object returned by the `mapSelect` function.

withDispatch

Higher-order component used to add dispatch props using registered action creators.

Usage

```
function Button( { onClick, children } ) {  
    return (  
        <button type="button" onClick={ onClick }>  
            { children }  
        </button>  
    );  
}  
  
import { withDispatch } from '@wordpress/data';  
import { store as myCustomStore } from 'my-custom-store';  
  
const SaleButton = withDispatch( ( dispatch, ownProps ) => {  
    const { startSale } = dispatch( myCustomStore );  
    const { discountPercent } = ownProps;  
  
    return {  
        onClick() {  
            startSale( discountPercent );  
        },  
    };  
} )( Button );  
  
// Rendered in the application:  
//  
// <SaleButton discountPercent="20">Start Sale!</SaleButton>
```

In the majority of cases, it will be sufficient to use only two first params passed to `mapDispatchToProps` as illustrated in the previous example. However, there might be some very advanced use cases where using the `registry` object might be used as a tool to optimize the performance of your component. Using `select` function from the registry might be useful when you need to fetch some dynamic data from the store at the time when the event is fired, but at the same time, you never use it to render your component. In such scenario, you can avoid using the `withSelect` higher order component to compute such prop, which might lead to unnecessary

re-renders of your component caused by its frequent value change. Keep in mind, that `mapDispatchToProps` must return an object with functions only.

```
function Button( { onClick, children } ) {
  return (
    <button type="button" onClick={ onClick }>
      { children }
    </button>
  );
}

import { withDispatch } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

const SaleButton = withDispatch( ( dispatch, ownProps, { select } ) => {
  // Stock number changes frequently.
  const { getStockNumber } = select( myCustomStore );
  const { startSale } = dispatch( myCustomStore );
  return {
    onClick() {
      const discountPercent = getStockNumber() > 50 ? 10 : 20;
      startSale( discountPercent );
    },
  };
})( Button );

// Rendered in the application:
// <SaleButton>Start Sale!</SaleButton>
```

Note: It is important that the `mapDispatchToProps` function always returns an object with the same keys. For example, it should not contain conditions under which a different value would be returned.

Parameters

- `mapDispatchToProps` Function: A function of returning an object of prop names where value is a dispatch-bound action creator, or a function to be called with the component's props and returning an action creator.

Returns

- `ComponentType`: Enhanced component with merged dispatcher props.

[withRegistry](#)

Higher-order component which renders the original component with the current registry context passed as its `registry` prop.

Parameters

- `OriginalComponent Component`: Original component.

Returns

- Component: Enhanced component.

withSelect

Higher-order component used to inject state-derived props using registered selectors.

Usage

```
import { withSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function PriceDisplay( { price, currency } ) {
    return new Intl.NumberFormat( 'en-US', {
        style: 'currency',
        currency,
    } ).format( price );
}

const HammerPriceDisplay = withSelect( ( select, ownProps ) => {
    const { getPrice } = select( myCustomStore );
    const { currency } = ownProps;

    return {
        price: getPrice( 'hammer', currency ),
    };
})( PriceDisplay );

// Rendered in the application:
// <HammerPriceDisplay currency="USD" />
```

In the above example, when HammerPriceDisplay is rendered into an application, it will pass the price into the underlying PriceDisplay component and update automatically if the price of a hammer ever changes in the store.

Parameters

- *mapSelectToProps* Function: Function called on every state change, expected to return object of props to merge with the component's own props.

Returns

- ComponentType: Enhanced component with merged state data props.

batch

As a response of `dispatch` calls, WordPress data based applications updates the connected components (Components using `useSelect` or `withSelect`). This update happens in two steps:

- The selectors are called with the update state.

- If the selectors return values that are different than the previous (strict equality), the component rerenders.

As the application grows, this can become costful, so it's important to ensure that we avoid running both these if possible. One of these situations happen when an interaction requires multiple consecutive `dispatch` calls in order to update the state properly. To avoid rerendering the components each time we call `dispatch`, we can wrap the sequential dispatch calls in `batch` which will ensure that the components only call selectors and rerender once at the end of the sequence.

Usage

```
import { useRegistry } from '@wordpress/data';

function Component() {
    const registry = useRegistry();

    function callback() {
        // This will only rerender the components once.
        registry.batch( () => {
            registry.dispatch( someStore ).someAction();
            registry.dispatch( someStore ).someOtherAction();
        } );
    }

    return <button onClick={ callback }>Click me</button>;
}


```

Selectors

The following selectors are available on the object returned by `wp.data.select('core')`.

Example

```
import { store as coreDataStore } from '@wordpress/core-data';
import { useSelect } from '@wordpress/data';

function Component() {
    const result = useSelect( ( select ) => {
        const query = { per_page: 20 };
        const selectorArgs = [ 'postType', 'page', query ];

        return {
            pages: select( coreDataStore ).getEntityRecords( ...selectorArgs ),
            hasStartedResolution: select( coreDataStore ).hasStartedResolution(
                'getEntityRecords', // _selectorName_
                selectorArgs
            ),
            hasFinishedResolution: select(
                coreDataStore
            ).hasFinishedResolution( 'getEntityRecords', selectorArgs ),
            isResolving: select( coreDataStore ).isResolving(
                'getEntityRecords',
            )
        };
    });
}
```

```

        selectorArgs
    ),
}
);

if ( result.hasStartedResolution ) {
    return <>Fetching data...</>;
}

if ( result.isResolving ) {
    return (
        <>
        {
            // show a spinner
        }
        </>
    );
}

if ( result.hasFinishedResolution ) {
    return (
        <>
        {
            // data is ready
        }
        </>
    );
}
}

```

[hasFinishedResolution](#)

Returns true if resolution has completed for a given selector name, and arguments set.

Parameters

- *state State*: Data state.
- *selectorName string*: Selector name.
- *args unknown[]?*: Arguments passed to selector.

Returns

- *boolean*: Whether resolution has completed.

[hasStartedResolution](#)

Returns true if resolution has already been triggered for a given selector name, and arguments set.

Parameters

- *state State*: Data state.
- *selectorName string*: Selector name.
- *args unknown[]?*: Arguments passed to selector.

Returns

- `boolean`: Whether resolution has been triggered.

isResolving

Returns true if resolution has been triggered but has not yet completed for a given selector name, and arguments set.

Parameters

- `state State`: Data state.
- `selectorName string`: Selector name.
- `args unknown[]?`: Arguments passed to selector.

Returns

- `boolean`: Whether resolution is in progress.

Normalizing Selector Arguments

In specific circumstances it may be necessary to normalize the arguments passed to a given *call* of a selector/resolver pairing.

Each resolver has [its resolution status cached in an internal state](#) where the [key is the arguments supplied to the selector](#) at *call* time.

For example for a selector with a single argument, the related resolver would generate a cache key of: [123].

[This cache is used to determine the resolution status of a given resolver](#) which is used to [avoid unwanted additional invocations of resolvers](#) (which often undertake “expensive” operations such as network requests).

As a result it’s important that arguments remain *consistent* when calling the selector. For example, by *default* these two calls will not be cached using the same key, even though they are likely identical:

```
// Arg as number  
getSomeDataById( 123 );  
  
// Arg as string  
getSomeDataById( '123' );
```

This is an opportunity to utilize the `__unstableNormalizeArgs` property to guarantee consistency by protecting callers from passing incorrect types.

Example

The 3rd argument of the following selector is intended to be a `Number`:

```
const getItemsSelector = ( name, type, id ) => {  
    return state.items[ name ][ type ][ id ] || null;  
};
```

However, it is possible that the `id` parameter will be passed as a `String`. In this case, the `__unstableNormalizeArgs` method (property) can be defined on the `selector` to coerce the arguments to the desired type even if they are provided “incorrectly”:

```
// Define normalization method.  
getItemsSelector.__unstableNormalizeArgs = ( args ) {  
    // "id" argument at the 2nd index  
    if (args[2] && typeof args[2] === 'string') {  
        args[2] === Number(args[2]);  
    }  
  
    return args;  
}
```

With this in place the following code will behave consistently:

```
const getItemsSelector = ( name, type, id ) => {  
    // here 'id' is now guaranteed to be a number.  
    return state.items[ name ][ type ][ id ] || null;  
};  
  
const getItemsResolver = ( name, type, id ) => {  
    // 'id' is also guaranteed to be a number in the resolver.  
    return {};  
};  
  
registry.registerStore( 'store', {  
    // ...  
    selectors: {  
        getItems: getItemsSelector,  
    },  
    resolvers: {  
        getItems: getItemsResolver,  
    },  
} );  
  
// Call with correct number type.  
registry.select( 'store' ).getItems( 'foo', 'bar', 54 );  
  
// Call with the wrong string type, **but** here we have avoided an  
// wanted resolver call because '54' is guaranteed to have been  
// coerced to a number by the `__unstableNormalizeArgs` method.  
registry.select( 'store' ).getItems( 'foo', 'bar', '54' );
```

Ensuring consistency of arguments for a given selector call is [an important optimization to help improve performance in the data layer](#). However, this type of problem can be usually be avoided by ensuring selectors don't use variable types for their arguments.

Going further

- [What is WordPress Data?](#)

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/data](#)

[Previous @wordpress/data-controls](#) [Previous: @wordpress/data-controls](#)
[Next @wordpress/dataviews](#) [Next: @wordpress/dataviews](#)

@wordpress/dataviews

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Data](#)
- [Pagination Info](#)
- [View](#)
 - [onChangeView: syncing view and data](#)
- [Fields](#)
- [Actions](#)
- [Types](#)
- [Other properties](#)
- [Contributing to this package](#)

[↑ Back to top](#)

DataViews is a component that provides an API to render datasets using different types of layouts (table, grid, list, etc.).

Installation

Install the module

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

Usage

```
<DataViews
  data={ data }
  paginationInfo={ { totalItems, totalPages } }
  view={ view }
  onChangeView={ onChangeView }
  fields={ fields }
  actions={ [ trashPostAction ] }
  search={ false }
  searchLabel="Filter list"
  getItemId={ ( item ) => item.id }
  isLoading={ isLoadingData }
  supportedLayouts={ [ 'table' ] }
  deferredRendering={ true }
  onSelectionChange={ ( items ) => { /* ... */ } }
/>
```

Data

The dataset to work with, represented as a one-dimensional array.

Example:

```
[  
  { id: 1, title: "Title", ... },  
  { ... }  
]
```

By default, dataviews would use each record's `id` as an unique identifier. If it's not, the consumer should provide a `getItemId` function that returns one. See “Other props” section.

Pagination Info

- `totalItems`: the total number of items in the datasets.
- `totalPages`: the total number of pages, taking into account the total items in the dataset and the number of items per page provided by the user.

View

The view object configures how the dataset is visible to the user.

Example:

```

{
  type: 'table',
  perPage: 5,
  page: 1,
  sort: {
    field: 'date',
    direction: 'desc',
  },
  search: '',
  filters: [
    { field: 'author', operator: 'in', value: 2 },
    { field: 'status', operator: 'in', value: 'publish,draft' }
  ],
  hiddenFields: [ 'date', 'featured-image' ],
  layout: {},
}

```

- `type`: view type, one of `table`, `grid`, `list`. See “View types”.
- `perPage`: number of records to show per page.
- `page`: the page that is visible.
- `sort.field`: field used for sorting the dataset.
- `sort.direction`: the direction to use for sorting, one of `asc` or `desc`.
- `search`: the text search applied to the dataset.
- `filters`: the filters applied to the dataset. Each item describes:
 - `field`: which field this filter is bound to.
 - `operator`: which type of filter it is. One of `in`, `notIn`. See “Operator types”.
 - `value`: the actual value selected by the user.
- `hiddenFields`: the `id` of the fields that are hidden in the UI.
- `layout`: config that is specific to a particular layout type.
 - `mediaField`: used by the `grid` and `list` layouts. The `id` of the field to be used for rendering each card’s media.
 - `primaryField`: used by the `grid` and `list` layouts. The `id` of the field to be highlighted in each card/list item.

onChangeView: syncing view and data

The view is a representation of the visible state of the dataset: what type of layout is used to display it (table, grid, etc.), how the dataset is filtered, how it is sorted or paginated.

It’s the consumer’s responsibility to work with the data provider to make sure the user options defined through the view’s config (sort, pagination, filters, etc.) are respected. The `onChangeView` prop allows the consumer to provide a callback to be called when the view config changes, to process the data accordingly.

The following example shows how a view object is used to query the WordPress REST API via the entities abstraction. The same can be done with any other data provider.

```

function MyCustomPageTable() {
  const [ view, setView ] = useState( {
    type: 'table',
    perPage: 5,
    page: 1,
    sort: {
      field: 'date',
    }
  })
}

```

```

        direction: 'desc',
    },
    search: '',
    filters: [
        { field: 'author', operator: 'in', value: 2 },
        { field: 'status', operator: 'in', value: 'publish,draft' }
    ],
    hiddenFields: [ 'date', 'featured-image' ],
    layout: {}
} );

const queryArgs = useMemo( () => {
    const filters = {};
    view.filters.forEach( ( filter ) => {
        if ( filter.field === 'status' && filter.operator === 'in' ) {
            filters.status = filter.value;
        }
        if ( filter.field === 'author' && filter.operator === 'in' ) {
            filters.author = filter.value;
        }
    } );
    return {
        per_page: view.perPage,
        page: view.page,
        _embed: 'author',
        order: view.sort?.direction,
        orderby: view.sort?.field,
        search: view.search,
        ...filters,
    };
}, [ view ] );

const {
    records
} = useEntityRecords( 'postType', 'page', queryArgs );

return (
    <DataViews
        data={ records }
        view={ view }
        onChangeView={ setView }
        ...
    />
);
}

```

Fields

The fields describe the visible items for each record in the dataset.

Example:

```

[
  {
    id: 'date',
    header: __( 'Date' ),
    getValue: ( { item } ) => item.date,
    render: ( { item } ) => {
      return (
        <time>{ getFormattedDate( item.date ) }</time>
      );
    },
    enableHiding: false
  },
  {
    id: 'author',
    header: __( 'Author' ),
    getValue: ( { item } ) => item.author,
    render: ( { item } ) => {
      return (
        <a href="...">{ item.author }</a>
      );
    },
    type: 'enumeration',
    elements: [
      { value: 1, label: 'Admin' }
      { value: 2, label: 'User' }
    ]
    enableSorting: false
  }
]

```

- **id**: identifier for the field. Unique.
- **header**: the field's name to be shown in the UI.
- **getValue**: function that returns the value of the field.
- **render**: function that renders the field.
- **elements**: the set of valid values for the field's value.
- **type**: the type of the field. Used to generate the proper filters. Only `enumeration` available at the moment. See “Field types”.
- **enableSorting**: whether the data can be sorted by the given field. True by default.
- **enableHiding**: whether the field can be hidden. True by default.
- **filterBy**: configuration for the filters.
 - **operators**: the list of operators supported by the field.

Actions

Array of operations that can be performed upon each record. Each action is an object with the following properties:

- **id**: string, required. Unique identifier of the action. For example, `move-to-trash`.
- **label**: string, required. User facing description of the action. For example, `Move to Trash`.
- **isPrimary**: boolean, optional. Whether the action should be listed inline (primary) or in hidden in the more actions menu (secondary).

- `icon`: icon to show for primary actions. It's required for a primary action, otherwise the action would be considered secondary.
- `isEligible`: function, optional. Whether the action can be performed for a given record. If not present, the action is considered to be eligible for all items. It takes the given record as input.
- `isDestructive`: boolean, optional. Whether the action can delete data, in which case the UI would communicate it via red color.
- `callback`: function, required unless `RenderModal` is provided. Callback function that takes the record as input and performs the required action.
- `RenderModal`: ReactElement, optional. If an action requires that some UI be rendered in a modal, it can provide a component which takes as props the record as `item` and a `closeModal` function. When this prop is provided, the `callback` property is ignored.
- `hideModalHeader`: boolean, optional. This property is used in combination with `RenderModal` and controls the visibility of the modal's header. If the action renders a modal and doesn't hide the header, the action's label is going to be used in the modal's header.

Types

- Layout types:
 - `table`: the view uses a table layout.
 - `grid`: the view uses a grid layout.
 - `list`: the view uses a list layout.
- Field types:
 - `enumeration`: the field value should be taken and can be filtered from a closed list of elements.
- Operator types:
 - `in`: operator to be used in filters for fields of type `enumeration`.
 - `notIn`: operator to be used in filters for fields of type `enumeration`.

Other properties

- `search`: whether the search input is enabled. `true` by default.
- `searchLabel`: what text to show in the search input. “Filter list” by default.
- `getItemId`: function that receives an item and returns an unique identifier for it. By default, it uses the `id` of the item as unique identifier. If it's not, the consumer should provide their own.
- `isLoading`: whether the data is loading. `false` by default.
- `supportedLayouts`: array of layouts supported. By default, all are: `table`, `grid`, `list`.
- `deferredRendering`: whether the items should be rendered asynchronously. Useful when there's a field that takes a lot of time (e.g.: previews). `false` by default.
- `onSelectionChange`: callback that signals the user selected one or more items, and takes them as parameter. So far, only the `list` view implements it.
- `onDetailsChange`: callback that signals the user triggered the details for one of more items, and takes them as parameter. So far, only the `list` view implements it.

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](#).

First published

December 2, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/dataviews”](#)

[Previous @wordpress/data](#) [Previous: @wordpress/data](#)
[Next @wordpress/date](#) [Next: @wordpress/date](#)

@wordpress/date

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [date](#)
 - [dateI18n](#)
 - [format](#)
 - [getDate](#)
 - [getSettings](#)
 - [gdate](#)
 - [gdateI18n](#)
 - [humanTimeDiff](#)
 - [isInTheFuture](#)
 - [setSettings](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Date module for WordPress.

Installation

Install the module

```
npm install @wordpress/date --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

date

Formats a date (like `date()` in PHP).

Related

- https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
- https://en.wikipedia.org/wiki/ISO_8601#Time_offsets_from_UTC

Parameters

- `dateFormat` `string`: PHP-style formatting string. See `php.net/date`.
- `dateValue` `Moment` | `Date` | `string` | `undefined`: Date object or string, parsable by `moment.js`.
- `timezone` `string` | `number` | `undefined`: Timezone to output result in or a UTC offset. Defaults to timezone from site.

Returns

- `string`: Formatted date in English.

dateI18n

Formats a date (like `wp_date()` in PHP), translating it into site's locale.

Backward Compatibility Notice: if `timezone` is set to `true`, the function behaves like `gmdateI18n`.

Related

- https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
- https://en.wikipedia.org/wiki/ISO_8601#Time_offsets_from_UTC

Parameters

- `dateFormat` `string`: PHP-style formatting string. See `php.net/date`.
- `dateValue` `Moment` | `Date` | `string` | `undefined`: Date object or string, parsable by `moment.js`.
- `timezone` `string` | `number` | `boolean` | `undefined`: Timezone to output result in or a UTC offset. Defaults to timezone from site. Notice: `boolean` is effectively deprecated, but still supported for backward compatibility reasons.

Returns

- `string`: Formatted date.

format

Formats a date. Does not alter the date's timezone.

Parameters

- *dateFormat* string: PHP-style formatting string. See [php.net/date](#).
- *dateValue* Moment | Date | string | undefined: Date object or string, parsable by moment.js.

Returns

- string: Formatted date.

getDate

Create and return a JavaScript Date Object from a date string in the WP timezone.

Parameters

- *dateString* string?: Date formatted in the WP timezone.

Returns

- Date: Date

getSettings

Returns the currently defined date settings.

Returns

- DateSettings: Settings, including locale data.

gmdate

Formats a date (like `date()` in PHP), in the UTC timezone.

Parameters

- *dateFormat* string: PHP-style formatting string. See [php.net/date](#).
- *dateValue* Moment | Date | string | undefined: Date object or string, parsable by moment.js.

Returns

- string: Formatted date in English.

gmdateI18n

Formats a date (like `wp_date()` in PHP), translating it into site's locale and using the UTC timezone.

Parameters

- *dateFormat* `string`: PHP-style formatting string. See [php.net/date](#).
- *dateValue* `Moment | Date | string | undefined`: Date object or string, parsable by moment.js.

Returns

- `string`: Formatted date.

[humanTimeDiff](#)

Returns a human-readable time difference between two dates, like [human_time_diff\(\)](#) in PHP.

Parameters

- *from* `Moment | Date | string`: From date, in the WP timezone.
- *to* `Moment | Date | string | undefined`: To date, formatted in the WP timezone.

Returns

- `string`: Human-readable time difference.

[isInTheFuture](#)

Check whether a date is considered in the future according to the WordPress settings.

Parameters

- *dateValue* `string`: Date String or Date object in the Defined WP Timezone.

Returns

- `boolean`: Is in the future.

[setSettings](#)

Adds a locale to moment, using the format supplied by `wp_localize_script()`.

Parameters

- *dateSettings* `DateSettings`: Settings, including locale data.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/date](#)

[Previous @wordpress/dataviews](#) Previous: [@wordpress/dataviews](#)

Next [@wordpress/dependency-extraction-webpack-plugin](#) Next: [@wordpress/dependency-extraction-webpack-plugin](#)

@wordpress/dependency-extraction-webpack-plugin

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
 - [Webpack](#)
 - [Behavior with scripts](#)
 - [Behavior with script modules](#)
 - [WordPress](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This webpack plugin serves two purposes:

- Externalize dependencies that are available as shared scripts or modules on WordPress sites.
- Add an asset file for each entry point that declares an object with the list of WordPress script or module dependencies for the entry point. The asset file also contains the current version calculated for the current source code.

This allows JavaScript bundles produced by webpack to leverage WordPress style dependency sharing without an error-prone process of manually maintaining a dependency list.

Version 5 of this plugin adds support for module bundling. [Webpack's output.module option](#) should be used to opt-in to this behavior. This plugin will adapt its behavior based on the `output.module` option, producing an asset file suitable for use with the WordPress Module API.

Consult the [webpack website](#) for additional information on webpack concepts.

Installation

Install the module

```
npm install @wordpress/dependency-extraction-webpack-plugin --save-dev
```

Note: This package requires Node.js 18.0.0 or later. It also requires webpack 5.0.0 or newer. It is not compatible with older versions.

Usage

Webpack

Use this plugin as you would other webpack plugins:

```
// webpack.config.js
const DependencyExtractionWebpackPlugin = require( '@wordpress/dependency-extraction-webpack-plugin' );

module.exports = {
    // ...snip
    plugins: [ new DependencyExtractionWebpackPlugin() ],
};
```

Note: Multiple instances of the plugin are not supported and may produce unexpected results. If you plan to extend the webpack configuration from `@wordpress/scripts` with your own `DependencyExtractionWebpackPlugin`, be sure to remove the default instance of the plugin:

```
const defaultConfig = require( '@wordpress/scripts/config/webpack.config' )
const webpackConfig = {
    ...defaultConfig,
    plugins: [
        ...defaultConfig.plugins.filter(
            ( plugin ) =>
                plugin.constructor.name !== 'DependencyExtractionWebpackPlugin'
        ),
        new DependencyExtractionWebpackPlugin( {
            injectPolyfill: true,
            requestToExternal( request ) {
                /* My externals */
            },
        } ),
    ],
};
```

Behavior with scripts

Each entry point in the webpack bundle will include an asset file that declares the WordPress script dependencies that should be enqueued. This file also contains the unique version hash calculated based on the file content.

For example:

```
// Source file entrypoint.js
import { Component } from 'react';

// Webpack will produce the output output/entrypoint.js
/* bundled JavaScript output */

// Webpack will also produce output/entrypoint.asset.php declaring script
<?php return array('dependencies' => array('react'), 'version' => 'dd4c2dc');
```

By default, the following module requests are handled:

Request	Global	Script handle
@babel/runtime/regenerator	regeneratorRuntime	wp-polyfill
@wordpress/*	wp['*']	wp-*
jquery	jQuery	jquery
lodash-es	lodash	lodash
lodash	lodash	lodash
moment	moment	moment
react-dom	ReactDOM	react-dom
react	React	react

Note: This plugin overlaps with the functionality provided by [webpack externals](#). This plugin is intended to extract script handles from bundle compilation so that a list of script dependencies does not need to be manually maintained. If you don't need to extract a list of script dependencies, use the `externals` option directly.

This plugin is compatible with `externals`, but they may conflict. For example, adding `{ externals: { '@wordpress/blob': 'wp.blob' } }` to webpack configuration will effectively hide the `@wordpress/blob` module from the plugin and it will not be included in dependency lists.

Behavior with script modules

Warning: Script modules support is considered experimental at this time.

This section describes the behavior of this package to bundle ECMAScript modules and generate asset files suitable for use with the WordPress Script Modules API.

Some of this plugin's options change, and webpack requires configuration to output script modules. Refer to [webpack's documentation](#) for up-to-date details.

```
const webpackConfig = {
  ...defaultConfig,

  // These lines are necessary to enable module compilation at time-of-w
  output: { module: true },
  experiments: { outputModule: true },

  plugins: [
    ...defaultConfig.plugins.filter(
```

```

        ( plugin ) =>
            plugin.constructor.name !== 'DependencyExtractionWebpackPlugin'
        ),
        new DependencyExtractionWebpackPlugin( {
            // With modules, use `requestToExternalModule`:
            requestToExternalModule( request ) {
                if ( request === 'my-registered-module' ) {
                    return request;
                }
            },
        } ),
    ],
};


```

Each entry point in the webpack bundle will include an asset file that declares the WordPress script module dependencies that should be enqueued. This file also contains the unique version hash calculated based on the file content.

For example:

```

// Source file entrypoint.js
import { store, getContext } from '@wordpress/interactivity';

// Webpack will produce the output output/entrypoint.js
/* bundled JavaScript output */

// Webpack will also produce output/entrypoint.asset.php declaring script
<?php return array('dependencies' => array('@wordpress/interactivity'), 'v

```

By default, the following script module requests are handled:

Request

@wordpress/interactivity

(@wordpress/interactivity is currently the only available WordPress script module.)

Note: This plugin overlaps with the functionality provided by [webpack externals](#). This plugin is intended to extract script module identifiers from bundle compilation so that a list of script module dependencies does not need to be manually maintained. If you don't need to extract a list of script module dependencies, use the `externals` option directly.

This plugin is compatible with `externals`, but they may conflict. For example, adding `{ externals: { '@wordpress/blob': 'wp.blob' } }` to webpack configuration will effectively hide the @wordpress/blob module from the plugin and it will not be included in dependency lists.

Options

An object can be passed to the constructor to customize the behavior, for example:

```

module.exports = {
    plugins: [
        new DependencyExtractionWebpackPlugin( { injectPolyfill: true } ),
    ],
};


```

```
],  
};  
  
outputFormat
```

- Type: string
- Default: php

The output format for the generated asset file. There are two options available: ‘php’ or ‘json’.

`outputFilename`

- Type: string | function
- Default: null

The filename for the generated asset file. Accepts the same values as the Webpack `output.filename` option.

`combineAssets`

- Type: boolean
- Default: false

By default, one asset file is created for each entry point. When this flag is set to true, all information about assets is combined into a single `assets.(json|php)` file generated in the output directory.

`combinedOutputFile`

- Type: string
- Default: null

This option is useful only when the `combineAssets` option is enabled. It allows providing a custom output file for the generated single assets file. It’s possible to provide a path that is relative to the output directory.

`useDefaults`

- Type: boolean
- Default: true

Pass `useDefaults: false` to disable the default request handling.

`injectPolyfill`

- Type: boolean
- Default: false

Force `wp-polyfill` to be included in each entry point’s dependency list. This would be the same as adding `import '@wordpress/polyfill';` to each entry point.

Note: This option is not available with script modules.

```
externalizedReport
```

- Type: boolean | string
- Default: false

Report all externalized dependencies as an array in JSON format. It could be used for further manual or automated inspection.

You can provide a filename, or set it to true to report to a default `externalized-dependencies.json`.

```
requestToExternal
```

Note: This option is not available with script modules. See [requestToExternalModule](#) for module usage.

- Type: function

`requestToExternal` allows the module handling to be customized. The function should accept a module request string and may return a string representing the global variable to use. An array of strings may be used to access globals via an object path, e.g. `wp.i18n` may be represented as `['wp', 'i18n']`.

`requestToExternal` provided via configuration has precedence over default external handling. Unhandled requests will be handled by the default unless `useDefaults` is set to `false`.

```
/**  
 * Externalize 'my-module'  
 *  
 * @param {string} request Requested module  
 *  
 * @return {(string|undefined)} Script global  
 */  
function requestToExternal( request ) {  
    // Handle imports like `import myModule from 'my-module'  
    if ( request === 'my-module' ) {  
        // Expect to find `my-module` as myModule in the global scope:  
        return 'myModule';  
    }  
}  
  
module.exports = {  
    plugins: [ new DependencyExtractionWebpackPlugin( { requestToExternal } );  
}
```

```
requestToExternalModule
```

Note: This option is only available with script modules. See [requestToExternal](#) for script usage.

- Type: function

`requestToExternalModule` allows the script module handling to be customized. The function should accept a script module request string and may return a string representing the script module to use. Often, the script module will have the same name.

`requestToExternalModule` provided via configuration has precedence over default external handling. Unhandled requests will be handled by the default unless `useDefaults` is set to `false`.

```
/**  
 * Externalize 'my-module'  
 *  
 * @param {string} request Requested script module  
 *  
 * @return {(string|boolean|undefined)} Script module ID  
 */  
function requestToExternalModule( request ) {  
    // Handle imports like `import myModule from 'my-module'  
    if ( request === 'my-module' ) {  
        // Import should be of the form `import { something } from "myModu  
        return 'myModule';  
    }  
  
    // If the script module ID in source is the same as the external scrip  
    return request === 'external-module-id-no-change-required';  
}  
  
module.exports = {  
    plugins: [  
        new DependencyExtractionWebpackPlugin( { requestToExternalModule }  
    ],  
};  
  
requestToHandle
```

Note: This option is not available with script modules. It has no corresponding module configuration.

- Type: function

All of the external modules handled by the plugin are expected to be WordPress script dependencies

and will be added to the dependency list. `requestToHandle` allows the script handle included in the dependency list to be customized.

If no string is returned, the script handle is assumed to be the same as the request.

`requestToHandle` provided via configuration has precedence over the defaults. Unhandled requests will be handled by the default unless `useDefaults` is set to `false`.

```
/**  
 * Map 'my-module' request to 'my-module-script-handle'  
 *  
 * @param {string} request Requested module  
 */
```

```

 * @return { (string|undefined) } Script global
 */
function requestToHandle( request ) {
    // Handle imports like `import myModule from 'my-module'`
    if ( request === 'my-module' ) {
        // `my-module` depends on the script with the 'my-module-script-handle'
        return 'my-module-script-handle';
    }
}

module.exports = {
    plugins: [ new DependencyExtractionWebpackPlugin( { requestToExternal
}];

requestToExternal and requestToHandle

```

The functions `requestToExternal` and `requestToHandle` allow this module to handle arbitrary modules.

`requestToExternal` is necessary to handle any module and maps a module request to a global name.

`requestToHandle` maps the same module request to a script handle, the strings that will be included in the `entrypoint.asset.php` files.

WordPress

Enqueue your script as usual and read the script dependencies dynamically:

```

$script_path      = 'path/to/script.js';
$script_asset_path = 'path/to/script.asset.php';
$script_asset     = file_exists( $script_asset_path )
    ? require( $script_asset_path )
    : array( 'dependencies' => array(), 'version' => filemtime( $script_path ) );
$script_url = plugins_url( $script_path, __FILE__ );
wp_enqueue_script( 'script', $script_url, $script_asset['dependencies'], $

```

Or with modules (the Script Module API is only available in WordPress > 6.5):

```

$module_path      = 'path/to/module.js';
$module_asset_path = 'path/to/module.asset.php';
$module_asset     = file_exists( $module_asset_path )
    ? require( $module_asset_path )
    : array( 'dependencies' => array(), 'version' => filemtime( $module_
$module_url = plugins_url( $module_path, __FILE__ );
wp_register_script_module( 'my-module', $module_url, $module_asset['depend
wp_enqueue_script_module( 'my-module' );

```

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/dependency-extraction-webpack-plugin”](#)

[Previous @wordpress/date](#) [Previous: @wordpress/date](#)

[Next @wordpress/deprecated](#) [Next: @wordpress/deprecated](#)

@wordpress/deprecated

In this article

[Table of Contents](#)

- [Installation](#)
- [Hook](#)
- [API](#)
 - [default](#)
 - [logged](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Deprecation utility for WordPress. Logs a message to notify developers about a deprecated feature.

Installation

Install the module

```
npm install @wordpress/deprecated --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.*

Hook

The `deprecated` action is fired with three parameters: the name of the deprecated feature, the options object passed to `deprecated`, and the message sent to the console.

Example:

```
import { addAction } from '@wordpress/hooks';

function addDeprecationAlert( message, { version } ) {
    alert( `Deprecation: ${ message }. Version: ${ version }` );
}

addAction(
    'deprecated',
    'my-plugin/add-deprecation-alert',
    addDeprecationAlert
);
```

API

default

Logs a message to notify developers about a deprecated feature.

Usage

```
import deprecated from '@wordpress/deprecated';

deprecated( 'Eating meat', {
    since: '2019.01.01',
    version: '2020.01.01',
    alternative: 'vegetables',
    plugin: 'the earth',
    hint: 'You may find it beneficial to transition gradually.',
} );
```

// Logs: 'Eating meat is deprecated since version 2019.01.01 and will be r

Parameters

- `feature` `string`: Name of the deprecated feature.
- `options` `[Object]`: Personalisation options
- `options.since` `[string]`: Version in which the feature was deprecated.
- `options.version` `[string]`: Version in which the feature will be removed.
- `options.alternative` `[string]`: Feature to use instead
- `options.plugin` `[string]`: Plugin name if it's a plugin feature
- `options.link` `[string]`: Link to documentation
- `options.hint` `[string]`: Additional message to help transition away from the deprecated feature.

logged

Object map tracking messages which have been logged, for use in ensuring a message is only logged once.

Type

- Record<string, true | undefined>

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/deprecated”](#)

[Previous @wordpress/dependency-extraction-webpack-plugin](#) Previous: [@wordpress/dependency-extraction-webpack-plugin](#)

[Next @wordpress/docgen](#) Next: [@wordpress/docgen](#)

(@)wordpress/docgen

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
 - [CLI options](#)
 - [Babel Configuration](#)
- [Examples](#)
 - [Default export](#)
 - [Named export](#)
 - [Namespace export](#)
 - [TypeScript support](#)
- [Contributing to this package](#)

[↑ Back to top](#)

`docgen` helps you to generate the *public API* of your code. Given an entry point file, it outputs the ES6 export statements and their corresponding JSDoc comments in human-readable format. It also supports TypeScript via the TypeScript babel plugin.

Some characteristics:

- If the export statement doesn't contain any JSDoc, it'll look up for JSDoc up to the declaration.
- It can resolve relative dependencies, either files or directories. For example, `import default from './dependency'` will find `dependency.js` or `dependency/index.js`
- For TypeScript support, all types must be explicitly annotated as the TypeScript Babel plugin is unable to consume inferred types (it does not run the TS compiler, after all—it merely parses TypeScript). For example, all function return types must be explicitly annotated if they are to be documented by `docgen`.

Installation

Install the module

```
npm install @wordpress/docgen --save-dev
```

Usage

```
npx docgen <entry-point.js>
```

This command will generate a file named `entry-point-api.md` containing all the exports and their JSDoc comments.

CLI options

- **--formatter (String)**: A path to a custom formatter to control the contents of the output file. It should be a CommonJS module that exports a function that takes as input:
 - `rootDir (String)`: current working directory as seen by `docgen`.
 - `docPath (String)`: path of the output document to generate.
 - `symbols (Array)`: the symbols found.
- **--ignore (RegExp)**: A regular expression used to ignore symbols whose name match it.
- **--output (String)**: Output file that will contain the API documentation.
- **--to-section (String)**: Append generated documentation to this section in the Markdown output. To be used by the default Markdown formatter. Depends on `--output` and bypasses the custom `--formatter` passed, if any.
- **--to-token**: Embed generated documentation within the start and end tokens in the Markdown output. To be used by the default Markdown formatter. Depends on `--output` and bypasses the custom `--formatter` passed, if any.
 - Start token: `<! -- START TOKEN(Autogenerated API docs) -->`
 - End token: `<! -- END TOKEN(Autogenerated API docs) -->`
- **--use-token (String)**: This option allows you to customize the string between the tokens. For example, `--use-token my-api` will look up for the start token `<! --`

- START TOKEN(my-api) --> and the end token <!-- END TOKEN(my-api) -->. Depends on --to-token.
- **--debug**: Run in debug mode, which outputs some intermediate files useful for debugging.

Babel Configuration

@wordpress/docgen follows the default [project-wide configuration of Babel](#). Like Babel, it will automatically search for a `babel.config.json` file, or an equivalent one using the [supported extensions](#), in the project root directory.

Without it, @wordpress/docgen runs with the default option. In other words, it cannot parse JSX or other advanced syntaxes.

Examples

Default export

Entry point `index.js`:

```
/** 
 * Adds two numbers.
 *
 * @param {number} term1 First number.
 * @param {number} term2 Second number.
 * @return {number} The result of adding the two numbers.
 */
export default function addition( term1, term2 ) {
    // Implementation would go here.
}
```

Output of `npx docgen index.js` would be `index-api.js`:

```
# API

## default

[example.js#L8-L10](example.js#L8-L10)

Adds two numbers.

**Parameters**

- **term1** `number`: First number.
- **term2** `number`: Second number.

**Returns**

`number` The result of adding the two numbers.
```

Named export

Entry point `index.js`:

```

/**
 * Adds two numbers.
 *
 * @param {number} term1 First number.
 * @param {number} term2 Second number.
 * @return {number} The result of adding the two numbers.
 */
function addition( term1, term2 ) {
    return term1 + term2;
}

/**
 * Adds two numbers.
 *
 * @deprecated Use `addition` instead.
 *
 * @param {number} term1 First number.
 * @param {number} term2 Second number.
 * @return {number} The result of adding the two numbers.
 */
function count( term1, term2 ) {
    return term1 + term2;
}

export { count, addition };

```

Output of `npx docgen index.js` would be `index-api.js`:

```

# API

## addition

[example.js#L25-L25](example.js#L25-L25)

Adds two numbers.

**Parameters**

- **term1** `number`: First number.
- **term2** `number`: Second number.

**Returns**

`number` The result of adding the two numbers.

## count

[example.js#L25-L25](example.js#L25-L25)

> **Deprecated** Use `addition` instead.

Adds two numbers.

**Parameters**

```

```

-  **term1** `number`: First number.
-  **term2** `number`: Second number.

**Returns**

`number` The result of adding the two numbers.

```

Namespace export

Let the entry point be `index.js`:

```
export * from './count';
```

with `./count/index.js` contents being:

```

/**
 * Subtracts two numbers.
 *
 * @example
 *
 * ```js
 * const result = subtraction( 5, 2 );
 * console.log( result ); // Will log 3
 * ```
 *
 * @param {number} term1 First number.
 * @param {number} term2 Second number.
 * @return {number} The result of subtracting the two numbers.
 */
export function subtraction( term1, term2 ) {
    return term1 - term2;
}

/**
 * Adds two numbers.
 *
 * @example
 *
 * ```js
 * const result = addition( 5, 2 );
 * console.log( result ); // Will log 7
 * ```
 *
 * @param {number} term1 First number.
 * @param {number} term2 Second number.
 * @return {number} The result of adding the two numbers.
 */
export function addition( term1, term2 ) {
    // Implementation would go here.
    return term1 + term2;
}

```

Output of `npx docgen index.js` would be `index-api.js`:

```

# API

## addition

[example-module.js#L1-L1](example-module.js#L1-L1)

Adds two numbers.

**Usage** 

```js
const result = addition(5, 2);
console.log(result); // Will log 7
```

**Parameters** 

- **term1** `number`: First number.
- **term2** `number`: Second number.

**Returns** 

`number` The result of adding the two numbers.

## subtraction

[example-module.js#L1-L1](example-module.js#L1-L1)

Subtracts two numbers.

**Usage** 

```js
const result = subtraction(5, 2);
console.log(result); // Will log 3
```

**Parameters** 

- **term1** `number`: First number.
- **term2** `number`: Second number.

**Returns** 

`number` The result of subtracting the two numbers.

```

TypeScript support

Entry point index.ts:

```

/**
 * Adds two numbers.
 *

```

```
* @param term1 First number.
* @param term2 Second number.
* @return The result of adding the two numbers.
*/
export default function addition( term1: number, term2: number ): number {
    // Implementation would go here.
}
```

Output of `npx docgen index.ts` would be `index-api.js`:

```
# API

## default

[example.js#L8-L10] (example.js#L8-L10)

Adds two numbers.

**Parameters**

- **term1** `number`: First number.
- **term2** `number`: Second number.

**Returns**

`number` The result of adding the two numbers.
```

[**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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/docgen”](#)

[Previous: @wordpress/deprecated](#) [Previous: @wordpress/deprecated](#)
[Next: @wordpress/dom-ready](#) [Next: @wordpress/dom-ready](#)

@wordpress/dom-ready

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [default](#)
- [Browser support](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Execute callback after the DOM is loaded.

[Installation](#)

Install the module

```
npm install @wordpress/dom-ready --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](#)

[default](#)

Specify a function to execute when the DOM is fully loaded.

Usage

```
import domReady from '@wordpress/dom-ready';

domReady( function () {
    //do something after DOM loads.
} );
```

Parameters

- **callback** **Callback:** A function to execute after the DOM is ready.

Returns

- **void:**

Browser support

See <https://make.wordpress.org/core/handbook/best-practices/browser-support/>

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/dom-ready”](#)

[Previous @wordpress/docgen](#) [Previous: @wordpress/docgen](#)
[Next @wordpress/dom](#) [Next: @wordpress/dom](#)

@wordpress/dom

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [computeCaretRect](#)
 - [documentHasSelection](#)
 - [documentHasTextSelection](#)
 - [documentHasUncollapsedSelection](#)
 - [focus](#)
 - [getFilesFromDataTransfer](#)
 - [getOffsetParent](#)
 - [getPhrasingContentSchema](#)
 - [getRectangleFromRange](#)
 - [getScrollContainer](#)
 - [insertAfter](#)
 - [isEmpty](#)
 - [isEntirelySelected](#)
 - [isFormElement](#)
 - [isHorizontalEdge](#)

- [isNumberInput](#)
- [isPhrasingContent](#)
- [isRTL](#)
- [isTextContent](#)
- [isTextField](#)
- [isVerticalEdge](#)
- [placeCaretAtHorizontalEdge](#)
- [placeCaretAtVerticalEdge](#)
- [remove](#)
- [removeInvalidHTML](#)
- [replace](#)
- [replaceTag](#)
- [safeHTML](#)
- [unwrap](#)
- [wrap](#)
- [Contributing to this package](#)

[↑ Back to top](#)

DOM utilities module for WordPress.

Installation

Install the module

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

API

computeCaretRect

Get the rectangle for the selection in a container.

Parameters

- *win Window*: The window of the selection.

Returns

- *DOMRect | null*: The rectangle.

documentHasSelection

Check whether the current document has a selection. This includes focus in input fields, textareas, and general rich-text selection.

Parameters

- *doc Document*: The document to check.

Returns

- `boolean`: True if there is selection, false if not.

[documentHasTextSelection](#)

Check whether the current document has selected text. This applies to ranges of text in the document, and not selection inside `<input>` and `<textarea>` elements.

See: https://developer.mozilla.org/en-US/docs/Web/API/Window/getSelection#Related_objects.

Parameters

- `doc Document`: The document to check.

Returns

- `boolean`: True if there is selection, false if not.

[documentHasUncollapsedSelection](#)

Check whether the current document has any sort of (uncollapsed) selection. This includes ranges of text across elements and any selection inside textual `<input>` and `<textarea>` elements.

Parameters

- `doc Document`: The document to check.

Returns

- `boolean`: Whether there is any recognizable text selection in the document.

[focus](#)

Object grouping `focusable` and `tabbable` utils under the keys with the same name.

[getFilesFromDataTransfer](#)

Gets all files from a DataTransfer object.

Parameters

- `dataTransfer DataTransfer`: DataTransfer object to inspect.

Returns

- `File[]`: An array containing all files.

[getOffsetParent](#)

Returns the closest positioned element, or null under any of the conditions of the offsetParent specification. Unlike offsetParent, this function is not limited to HTMLElement and accepts any Node (e.g. `Node.TEXT_NODE`).

Related

- <https://drafts.csswg.org/cssom-view/#dom-htmlelement-offsetparent>

Parameters

- `node Node`: Node from which to find offset parent.

Returns

- `Node | null`: Offset parent.

[**getPhrasingContentSchema**](#)

Get schema of possible paths for phrasing content.

Related

- https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_categories#Phrasing_content

Parameters

- `context [string]`: Set to “paste” to exclude invisible elements and sensitive data.

Returns

- `Partial<ContentSchema>`: Schema.

[**getRectangleFromRange**](#)

Get the rectangle of a given Range. Returns `null` if no suitable rectangle can be found.

Parameters

- `range Range`: The range.

Returns

- `DOMRect?`: The rectangle.

[**getScrollContainer**](#)

Given a DOM node, finds the closest scrollable container node or the node itself, if scrollable.

Parameters

- `node Element | null`: Node from which to start.
- `direction ?string`: Direction of scrollable container to search for (‘vertical’, ‘horizontal’, ‘all’). Defaults to ‘vertical’.

Returns

- `Element | undefined`: Scrollable container node, if found.

[insertAfter](#)

Given two DOM nodes, inserts the former in the DOM as the next sibling of the latter.

Parameters

- *newNode* Node: Node to be inserted.
- *referenceNode* Node: Node after which to perform the insertion.

Returns

- void:

[isEmpty](#)

Recursively checks if an element is empty. An element is not empty if it contains text or contains elements with attributes such as images.

Parameters

- *element* Element: The element to check.

Returns

- boolean: Whether or not the element is empty.

[isEntirelySelected](#)

Check whether the contents of the element have been entirely selected. Returns true if there is no possibility of selection.

Parameters

- *element* HTMLElement: The element to check.

Returns

- boolean: True if entirely selected, false if not.

[isFormElement](#)

Detects if element is a form element.

Parameters

- *element* Element: The element to check.

Returns

- boolean: True if form element and false otherwise.

[isHorizontalEdge](#)

Check whether the selection is horizontally at the edge of the container.

Parameters

- *container* `HTMLElement`: Focusable element.
- *isReverse* `boolean`: Set to true to check left, false for right.

Returns

- `boolean`: True if at the horizontal edge, false if not.

[isNumberInput](#)

Check whether the given element is an input field of type number.

Parameters

- *node* `Node`: The HTML node.

Returns

- *node* `is HTMLInputElement`: True if the node is number input.

[isPhrasingContent](#)

Find out whether or not the given node is phrasing content.

Related

- https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_categories#Phrasing_content

Parameters

- *node* `Node`: The node to test.

Returns

- `boolean`: True if phrasing content, false if not.

[isRTL](#)

Whether the element's text direction is right-to-left.

Parameters

- *element* `Element`: The element to check.

Returns

- `boolean`: True if rtl, false if ltr.

[isTextContent](#)

Parameters

- *node* `Node`:

Returns

- `boolean`: Node is text content

[isTextField](#)

Check whether the given element is a text field, where text field is defined by the ability to select within the input, or that it is contenteditable.

See: <https://html.spec.whatwg.org/#textFieldSelection>

Parameters

- `node Node`: The HTML element.

Returns

- `node is HTMLElement`: True if the element is an text field, false if not.

[isVerticalEdge](#)

Check whether the selection is vertically at the edge of the container.

Parameters

- `container HTMLElement`: Focusable element.
- `isReverse boolean`: Set to true to check top, false for bottom.

Returns

- `boolean`: True if at the vertical edge, false if not.

[placeCaretAtHorizontalEdge](#)

Places the caret at start or end of a given element.

Parameters

- `container HTMLElement`: Focusable element.
- `isReverse boolean`: True for end, false for start.

[placeCaretAtVerticalEdge](#)

Places the caret at the top or bottom of a given element.

Parameters

- `container HTMLElement`: Focusable element.
- `isReverse boolean`: True for bottom, false for top.
- `rect [DOMRect]`: The rectangle to position the caret with.

[remove](#)

Given a DOM node, removes it from the DOM.

Parameters

- *node* `Node`: Node to be removed.

Returns

- `void`:

[removeInvalidHTML](#)

Given a schema, unwraps or removes nodes, attributes and classes on HTML.

Parameters

- *HTML string*: The HTML to clean up.
- *schema import('./clean-node-list')*.`Schema`: Schema for the HTML.
- *inline boolean*: Whether to clean for inline mode.

Returns

- `string`: The cleaned up HTML.

[replace](#)

Given two DOM nodes, replaces the former with the latter in the DOM.

Parameters

- *processedNode* `Element`: Node to be removed.
- *newNode* `Element`: Node to be inserted in its place.

Returns

- `void`:

[replaceTag](#)

Replaces the given node with a new node with the given tag name.

Parameters

- *node* `Element`: The node to replace
- *tagName* `string`: The new tag name.

Returns

- `Element`: The new node.

[safeHTML](#)

Strips scripts and on* attributes from HTML.

Parameters

- *html* `string`: HTML to sanitize.

Returns

- `string`: The sanitized HTML.

[unwrap](#)

Unwrap the given node. This means any child nodes are moved to the parent.

Parameters

- `node Node`: The node to unwrap.

Returns

- `void`:

[wrap](#)

Wraps the given node with a new node with the given tag name.

Parameters

- `newNode Element`: The node to insert.
- `referenceNode Element`: The node to wrap.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/dom”](#)

[Previous: @wordpress/dom-ready](#) [Previous: @wordpress/dom-ready](#)

[Next: @wordpress/e2e-test-utils-playwright](#) [Next: @wordpress/e2e-test-utils-playwright](#)

@wordpress/e2e-test-utils-playwright

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [test](#)
 - [expect](#)
 - [Admin](#)
 - [Editor](#)
 - [PageUtils](#)
 - [RequestUtils](#)
- [Contributing to this package](#)

[↑ Back to top](#)

End-To-End (E2E) Playwright test utils for WordPress.

It works properly with the minimum version of Gutenberg 9.2.0 or the minimum version of WordPress 5.6.0.

This package is still under active development. Documentation might not be up-to-date, and the v0.x version can introduce breaking changes without a detailed migration guide. Early adopters are encouraged to use a [lock file](#) to prevent unexpected breakages.

[Installation](#)

Install the module

```
npm install @wordpress/e2e-test-utils-playwright --save-dev
```

Note: This package requires Node.js 12.0.0 or later. It is not compatible with older versions.

[API](#)

[test](#)

The extended Playwright's [test](#) module with the `admin`, `editor`, `pageUtils` and the `requestUtils` fixtures.

[expect](#)

The Playwright/Jest's [expect](#) function.

[Admin](#)

End to end test utilities for WordPress admin's user interface.

```
const admin = new Admin( { page, pageUtils } );
await admin.visitAdminPage( 'options-general.php' );
```

[Editor](#)

End to end test utilities for the WordPress Block Editor.

To use these utilities, instantiate them within each test file:

```
test.use( {
  editor: async ( { page }, use ) => {
    await use( new Editor( { page } ) );
  },
} );
```

Within a test or test utility, use the `canvas` property to select elements within the iframe canvas:

```
await editor.canvas.locator( 'role=document[name="Paragraph block"]i' )
```

[PageUtils](#)

Generic Playwright utilities for interacting with web pages.

```
const pageUtils = new PageUtils( { page } );
await pageUtils.pressKeys( 'primary+a' );
```

[RequestUtils](#)

Playwright utilities for interacting with the WordPress REST API.

Create a request utils instance.

```
const requestUtils = await RequestUtils.setup( {
  user: {
    username: 'admin',
    password: 'password',
  },
} );
```

[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](#).

First published

May 5, 2023

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: @wordpress/e2e-test-utils-playwright”](#)

[Previous @wordpress/dom](#) [Previous: @wordpress/dom](#)

[Next @wordpress/e2e-test-utils](#) [Next: @wordpress/e2e-test-utils](#)

@wordpress/e2e-test-utils

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [activatePlugin](#)
 - [activateTheme](#)
 - [arePrePublishChecksEnabled](#)
 - [canvas](#)
 - [changeSiteTimezone](#)
 - [clearLocalStorage](#)
 - [clickBlockAppender](#)
 - [clickBlockToolbarButton](#)
 - [clickButton](#)
 - [clickMenuItem](#)
 - [clickOnCloseModalButton](#)
 - [clickOnMoreMenuItem](#)
 - [closeGlobalBlockInserter](#)
 - [closeListView](#)
 - [createEmbeddingMatcher](#)
 - [createJSONResponse](#)
 - [createMenu](#)
 - [createNewPost](#)
 - [createNewTemplate](#)
 - [createReusableBlock](#)
 - [createUrl](#)
 - [createUrlMatcher](#)
 - [createUser](#)
 - [deactivatePlugin](#)
 - [deleteAllMenus](#)
 - [deleteAllTemplates](#)
 - [deleteAllWidgets](#)
 - [deleteTheme](#)
 - [deleteUser](#)
 - [disableFocusLossObservation](#)
 - [disablePageDialogAccept](#)
 - [disablePrePublishChecks](#)
 - [disableSiteEditorWelcomeGuide](#)
 - [dragAndResize](#)
 - [enableFocusLossObservation](#)

- [enablePageDialogAccept](#)
- [enablePrePublishChecks](#)
- [ensureSidebarOpened](#)
- [enterEditMode](#)
- [findSidebarPanelToggleButtonWithTitle](#)
- [findSidebarPanelWithTitle](#)
- [getAllBlockInserterItemTitles](#)
- [getAllBlocks](#)
- [getAvailableBlockTransforms](#)
- [getBlockSetting](#)
- [getCurrentPostContent](#)
- [getCurrentSiteEditorContent](#)
- [getEditedPostContent](#)
- [getListViewById](#)
- [getOption](#)
- [getPageError](#)
- [hasBlockSwitcher](#)
- [insertBlock](#)
- [insertBlockDirectoryBlock](#)
- [insertPattern](#)
- [installPlugin](#)
- [installTheme](#)
- [isCurrentURL](#)
- [isDefaultBlock](#)
- [isListViewOpen](#)
- [isOfflineMode](#)
- [isThemeInstalled](#)
- [loginUser](#)
- [logout](#)
- [mockOrTransform](#)
- [openDocumentSettingsSidebar](#)
- [openGlobalBlockInserter](#)
- [openGlobalStylesPanel](#)
- [openListView](#)
- [openPreviewPage](#)
- [openPreviousGlobalStylesPanel](#)
- [openPublishPanel](#)
- [openTypographyToolsPanelMenu](#)
- [pressKeyTimes](#)
- [pressKeyWithModifier](#)
- [publishPost](#)
- [publishPostWithPrePublishChecksDisabled](#)
- [resetPreferences](#)
- [saveDraft](#)
- [searchForBlock](#)
- [searchForBlockDirectoryBlock](#)
- [searchForPattern](#)
- [searchForReusableBlock](#)
- [selectBlockByClientId](#)
- [setBrowserViewport](#)
- [setClipboardData](#)
- [setOption](#)
- [setPostContent](#)
- [setUpResponseMocking](#)

- [showBlockToolbar](#)
 - [switchBlockInspectorTab](#)
 - [switchEditorModeTo](#)
 - [switchUserToAdmin](#)
 - [switchUserToTest](#)
 - [toggleGlobalBlockInserter](#)
 - [toggleGlobalStyles](#)
 - [toggleMoreMenu](#)
 - [toggleOfflineMode](#)
 - [togglePreferencesOption](#)
 - [transformBlockTo](#)
 - [trashAllComments](#)
 - [trashAllPosts](#)
 - [uninstallPlugin](#)
 - [visitAdminPage](#)
 - [visitSiteEditor](#)
 - [waitForWindowDimensions](#)
 - [wpDataSelect](#)
- [Contributing to this package](#)

[↑ Back to top](#)

End-To-End (E2E) test utils for WordPress.

It works properly with the minimum version of Gutenberg 13.8.0 or the minimum version of WordPress 6.0.0.

Note that there's currently an ongoing [project](#) to migrate E2E tests to Playwright instead. This package is deprecated and will only accept bug fixes until fully migrated.

Installation

Install the module

```
npm install @wordpress/e2e-test-utils --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

API

activatePlugin

Activates an installed plugin.

Parameters

- `slug` `string`: Plugin slug.

activateTheme

Activates an installed theme.

Parameters

- *slug* `string`: Theme slug.

arePrePublishChecksEnabled

Verifies if publish checks are enabled.

Returns

- `Promise<boolean>`: Boolean which represents the state of prepublish checks.

canvas

Gets the editor canvas frame.

changeSiteTimezone

Visits general settings page and changes the timezone to the given value.

Parameters

- *timezone* `string`: Value of the timezone to set.

Returns

- `string`: Value of the previous timezone.

clearLocalStorage

Clears the local storage.

clickBlockAppender

Clicks the default block appender.

clickBlockToolbarButton

Clicks a block toolbar button.

Parameters

- *label* `string`: The text string of the button label.
- *type* `[string]`: The type of button label: ‘ariaLabel’ or ‘content’.

clickButton

Clicks a button based on the text on the button.

Parameters

- *buttonText* `string`: The text that appears on the button to click.

[clickMenuItem](#)

Searches for an item in the menu with the text provided and clicks it.

Parameters

- *label* `string`: The label to search the menu item for.

[clickOnCloseModalButton](#)

Click on the close button of an open modal.

Parameters

- *modalClassName* `?string`: Class name for the modal to close

[clickOnMoreMenuItem](#)

Clicks on More Menu item, searches for the button with the text provided and clicks it.

Parameters

- *buttonLabel* `string`: The label to search the button for.

[closeGlobalBlockInserter](#)

Closes the global inserter.

[closeListView](#)

Closes list view

[createEmbeddingMatcher](#)

Creates a function to determine if a request is embedding a certain URL.

Parameters

- *url* `string`: The URL to check against a request.

Returns

- `Function`: Function that determines if a request is for the embed API, embedding a specific URL.

[createJSONResponse](#)

Respond to a request with a JSON response.

Parameters

- *mockResponse* `string`: The mock object to wrap in a JSON response.

Returns

- **Promise**: Promise that responds to a request with the mock JSON response.

[createMenu](#)

Create menus and all linked resources for the menu using the REST API.

Parameters

- *menu Object*: Rest payload for the menu
- *menuItems ?Array*: Data for any menu items to be created.

[createNewPost](#)

Creates new post.

Parameters

- *object Object*: Object to create new post, along with tips enabling option.
- *object.postType [string]*: Post type of the new post.
- *object.title [string]*: Title of the new post.
- *object.content [string]*: Content of the new post.
- *object.excerpt [string]*: Excerpt of the new post.
- *object.showWelcomeGuide [boolean]*: Whether to show the welcome guide.

[createNewTemplate](#)

Opens the template editor with a newly created template.

Parameters

- *name string*: Name of the template.

[createReusableBlock](#)

Creates a simple reusable block with a paragraph block.

Parameters

- *content string*: Paragraph block's content
- *title title*: Reusable block's name.

[createUrl](#)

Creates new URL by parsing base URL, WPPath and query string.

Parameters

- *WPPath string*: String to be serialized as pathname.
- *query ?string*: String to be serialized as query portion of URL.

Returns

- **string**: String which represents full URL.

createUrlMatcher

Creates a function to determine if a request is calling a URL with the substring present.

Parameters

- *substring string*: The substring to check for.

Returns

- **Function**: Function that determines if a request's URL contains substring.

createUser

Create a new user account.

Parameters

- *username string*: User name.
- *object Object?*: Optional Settings for the new user account.
- *object.firstName [string]*: First name.
- *object.lastName [string]*: Last name.
- *object.role [string]*: Role. Defaults to Administrator.

Returns

- **string**: Password for the newly created user account.

deactivatePlugin

Deactivates an active plugin.

Parameters

- *slug string*: Plugin slug.

deleteAllMenus

Delete all menus using the REST API

deleteAllTemplates

Delete all the templates of given type.

Parameters

- *type ('wp_template' | 'wp_template_part')*:- Template type to delete.

[deleteAllWidgets](#)

Delete all the widgets in the widgets screen.

[deleteTheme](#)

Deletes a theme from the site, activating another theme if necessary.

Parameters

- *slug* `string`: Theme slug.
- *settings* `Object?`: Optional settings object.
- *settings.newThemeSlug* `string?`: A theme to switch to if the theme to delete is active. Required if the theme to delete is active.
- *settings.newThemeSearchTerm* `string?`: A search term to use if the new theme is not findable by its slug.

[deleteUser](#)

Delete a user account.

Parameters

- *username* `string`: User name.

[disableFocusLossObservation](#)

Removes the focus loss listener that `enableFocusLossObservation()` adds.

[disablePageDialogAccept](#)

Disable auto-accepting any dialogs.

[disablePrePublishChecks](#)

Disables Pre-publish checks.

[disableSiteEditorWelcomeGuide](#)

Skips the welcome guide popping up to first time users of the site editor

[dragAndResize](#)

Clicks an element, drags a particular distance and releases the mouse button.

Parameters

- *element* `Object`: The puppeteer element handle.
- *delta* `Object`: Object containing movement distances.
- *delta.x* `number`: Horizontal distance to drag.
- *delta.y* `number`: Vertical distance to drag.

Returns

- `Promise`: Promise resolving when drag completes.

[enableFocusLossObservation](#)

Adds an event listener to the document which throws an error if there is a loss of focus.

[enablePageDialogAccept](#)

Enables event listener which auto-accepts all dialogs on the page.

[enablePrePublishChecks](#)

Enables Pre-publish checks.

[ensureSidebarOpened](#)

Verifies that the edit post/site/widgets sidebar is opened, and if it is not, opens it.

Returns

- `Promise`: Promise resolving once the sidebar is opened.

[enterEditMode](#)

Enters edit mode.

[findSidebarPanelToggleButtonWithTitle](#)

Finds a sidebar panel with the provided title.

Parameters

- `panelTitle string`: The name of sidebar panel.

Returns

- `?ElementHandle`: Object that represents an in-page DOM element.

[findSidebarPanelWithTitle](#)

Finds the button responsible for toggling the sidebar panel with the provided title.

Parameters

- `panelTitle string`: The name of sidebar panel.

Returns

- `Promise<ElementHandle|undefined>`: Object that represents an in-page DOM element.

[getAllBlockInserterItemTitles](#)

Returns an array of strings with all inserter item titles.

Returns

- **Promise**: Promise resolving with an array containing all inserter item titles.

[getAllBlocks](#)

Returns an array with all blocks; Equivalent to calling wp.data.select('core/block-editor').getBlocks();

Returns

- **Promise**: Promise resolving with an array containing all blocks in the document.

[getAvailableBlockTransforms](#)

Returns an array of strings with all block titles, that the current selected block can be transformed into.

Returns

- **Promise**: Promise resolving with an array containing all possible block transforms

[getBlockSetting](#)

Returns a string containing the block title associated with the provided block name.

Parameters

- *blockName* **string**: Block name.
- *setting* **string**: Block setting e.g: title, attributes....

Returns

- **Promise**: Promise resolving with a string containing the block title.

[getCurrentPostContent](#)

Returns a promise which resolves with the current post content (HTML string).

Returns

- **Promise**: Promise resolving with current post content markup.

[getCurrentSiteEditorContent](#)

Returns a promise which resolves with the edited post content (HTML string).

Returns

- **Promise<string>**: Promise resolving with post content markup.

[getEditedPostContent](#)

Returns a promise which resolves with the edited post content (HTML string).

Returns

- **Promise**: Promise resolving with post content markup.

[getListviewBlocks](#)

Gets all block anchor nodes in the list view that match a given block name label.

Parameters

- *blockLabel* **string**: the label of the block as displayed in the ListView.

Returns

- **Promise**: all the blocks anchor nodes matching the label in the ListView.

[getOption](#)

Returns a site option, from the options admin page.

Parameters

- *setting* **string**: The option, used to get the option by id.

Returns

- **string**: The value of the option.

[getPageError](#)

Returns a promise resolving to one of either a string or null. A string will be resolved if an error message is present in the contents of the page. If no error is present, a null value will be resolved instead. This requires the environment be configured to display errors.

Related

- <http://php.net/manual/en/function.error-reporting.php>

Returns

- **Promise<?string>**: Promise resolving to a string or null, depending whether a page error is present.

[hasBlockSwitcher](#)

Returns a boolean indicating if the current selected block has a block switcher or not.

Returns

- **Promise**: Promise resolving with a boolean.

[insertBlock](#)

Inserts a block matching a given search term via the global inserter.

Parameters

- *searchTerm* `string`: The term by which to find the block to insert.

[insertBlockDirectoryBlock](#)

Inserts a Block Directory block matching a given search term via the global inserter.

Parameters

- *searchTerm* `string`: The term by which to find the Block Directory block to insert.

[insertPattern](#)

Inserts a pattern matching a given search term via the global inserter.

Parameters

- *searchTerm* `string`: The term by which to find the pattern to insert.

[installPlugin](#)

Installs a plugin from the WP.org repository.

Parameters

- *slug* `string`: Plugin slug.
- *searchTerm* `string?`: If the plugin is not findable by its slug use an alternative term to search.

[installTheme](#)

Installs a theme from the WP.org repository.

Parameters

- *slug* `string`: Theme slug.
- *settings* `Object?`: Optional settings object.
- *settings.searchTerm* `string?`: Search term to use if the theme is not findable by its slug.

[isCurrentURL](#)

Checks if current URL is a WordPress path.

Parameters

- *WPPath* `string`: String to be serialized as pathname.
- *query* `?string`: String to be serialized as query portion of URL.

Returns

- `boolean`: Boolean represents whether current URL is or not a WordPress path.

[isInDefaultBlock](#)

Checks if the block that is focused is the default block.

Returns

- `Promise`: Promise resolving with a boolean indicating if the focused block is the default block.

[isListViewOpen](#)

Undocumented declaration.

[isOfflineMode](#)

Undocumented declaration.

[isThemeInstalled](#)

Checks whether a theme exists on the site.

Parameters

- `slug string`: Theme slug to check.

Returns

- `boolean`: Whether the theme exists.

[loginUser](#)

Performs log in with specified username and password.

Parameters

- `username ?string`: String to be used as user credential.
- `password ?string`: String to be used as user credential.

[logout](#)

Performs log out.

[mockOrTransform](#)

Mocks a request with the supplied mock object, or allows it to run with an optional transform, based on the deserialised JSON response for the request.

Parameters

- *mockCheck Function*: function that returns true if the request should be mocked.
- *mock Object*: A mock object to wrap in a JSON response, if the request should be mocked.
- *responseObjectTransform Function|undefined*: An optional function that transforms the response's object before the response is used.

Returns

- **Promise**: Promise that uses `mockCheck` to see if a request should be mocked with `mock`, and optionally transforms the response with `responseObjectTransform`.

[openDocumentSettingsSidebar](#)

Clicks on the button in the header which opens Document Settings sidebar when it is closed.

[openGlobalBlockInserter](#)

Opens the global inserter.

[openGlobalStylesPanel](#)

Opens a global styles panel.

Parameters

- *panelName string*: Name of the panel that is going to be opened.

[openListView](#)

Opens list view

[openPreviewPage](#)

Opens the preview page of an edited post.

Parameters

- *editorPage Page*: puppeteer editor page.

Returns

- **Page**: preview page.

[openPreviousGlobalStylesPanel](#)

Opens the previous global styles panel.

[openPublishPanel](#)

Opens the publish panel.

[openTypographyToolsPanelMenu](#)

Opens the Typography tools panel menu provided via block supports.

[pressKeyTimes](#)

Presses the given keyboard key a number of times in sequence.

Parameters

- *key string*: Key to press.
- *count number*: Number of times to press.

[pressKeyWithModifier](#)

Performs a key press with modifier (Shift, Control, Meta, Alt), where each modifier is normalized to platform-specific modifier.

Parameters

- *modifier string*: Modifier key.
- *key string*: Key to press while modifier held.

[publishPost](#)

Publishes the post, resolving once the request is complete (once a notice is displayed).

Returns

- `Promise`: Promise resolving when publish is complete.

[publishPostWithPrePublishChecksDisabled](#)

Publishes the post without the pre-publish checks, resolving once the request is complete (once a notice is displayed).

Returns

- `Promise`: Promise resolving when publish is complete.

[resetPreferences](#)

Clears all user meta preferences.

[saveDraft](#)

Saves the post as a draft, resolving once the request is complete (once the “Saved” indicator is displayed).

Returns

- `Promise`: Promise resolving when draft save is complete.

[searchForBlock](#)

Searches for a block via the global inserter.

Parameters

- *searchTerm* `string`: The term to search the inserter for.

Returns

- `Promise<ElementHandle|null>`: The handle of block to be inserted or null if nothing was found.

[searchForBlockDirectoryBlock](#)

Searches for a Block Directory block via the global inserter.

Parameters

- *searchTerm* `string`: The term to search the inserter for.

Returns

- `Promise<ElementHandle|null>`: The handle of the Block Directory block to be inserted or null if nothing was found.

[searchForPattern](#)

Searches for a pattern via the global inserter.

Parameters

- *searchTerm* `string`: The term to search the inserter for.

Returns

- `Promise<ElementHandle|null>`: The handle of the pattern to be inserted or null if nothing was found.

[searchForReusableBlock](#)

Searches for a reusable block via the global inserter.

Parameters

- *searchTerm* `string`: The term to search the inserter for.

Returns

- `Promise<ElementHandle|null>`: The handle of the reusable block to be inserted or null if nothing was found.

[selectBlockByClientId](#)

Given the clientId of a block, selects the block on the editor.

Parameters

- *clientId* string: Identified of the block.

[setBrowserViewport](#)

Sets browser viewport to specified type.

Parameters

- *viewport* WPViewport: Viewport name or dimensions object to assign.

[setClipboardData](#)

Sets the clipboard data that can be pasted with `pressKeyWithModifier('primary' , 'v')`.

Parameters

- *\$1 Object*: Options.
- *\$1.plainText* string: Plain text to set.
- *\$1.html* string: HTML to set.

[setOption](#)

Sets a site option, from the options-general admin page.

Parameters

- *setting* string: The option, used to get the option by id.
- *value* string: The value to set the option to.

Returns

- string: The previous value of the option.

[setPostContent](#)

Sets code editor content

Parameters

- *content* string: New code editor content.

Returns

- Promise: Promise resolving with an array containing all blocks in the document.

[setUpResponseMocking](#)

Sets up mock checks and responses. Accepts a list of mock settings with the following properties:

- `match`: function to check if a request should be mocked.
- `onRequestMatch`: async function to respond to the request.

Usage

```
const MOCK_RESPONSES = [
  {
    match: isEmbedding( 'https://wordpress.org/gutenberg/handbook/' ),
    onRequestMatch: JSONResponse( MOCK_BAD_WORDPRESS_RESPONSE ),
  },
  {
    match: isEmbedding(
      'https://wordpress.org/gutenberg/handbook/block-api/attributes'
    ),
    onRequestMatch: JSONResponse( MOCK_EMBED_WORDPRESS_SUCCESS_RESPONSE )
  },
];
setUpResponseMocking( MOCK_RESPONSES );
```

If none of the mock settings match the request, the request is allowed to continue.

Parameters

- `mocks Array`: Array of mock settings.

[showBlockToolbar](#)

The block toolbar is not always visible while typing. Call this function to reveal it.

[switchBlockInspectorTab](#)

Clicks on the block inspector tab button with the supplied label and waits for the tab switch.

Parameters

- `label string`: Aria label to find tab button by.

[switchEditorModeTo](#)

Switches editor mode.

Parameters

- `mode string`: String editor mode.

[switchUserToAdmin](#)

Switches the current user to the admin user (if the user running the test is not already the admin user).

[switchUserToTest](#)

Switches the current user to whichever user we should be running the tests as (if we're not already that user).

[toggleGlobalBlockInserter](#)

Toggles the global inserter.

[toggleGlobalStyles](#)

Toggles the global styles sidebar (opens it if closed and closes it if open).

[toggleMoreMenu](#)

Toggles the More Menu.

Parameters

- *waitFor* ['open' | 'close']: Whether it should wait for the menu to open or close. If undefined it won't wait for anything.

[toggleOfflineMode](#)

Undocumented declaration.

[togglePreferencesOption](#)

Toggles a preference option with the given tab label and the option label.

Parameters

- *tabLabel* string: The preferences tab label to click.
- *optionLabel* string: The option label to search the button for.
- *shouldBeChecked* [boolean]: If true, turns the option on. If false, off. If not provided, the option will be toggled.

[transformBlockTo](#)

Converts editor's block type.

Parameters

- *name* string: Block name.

[trashAllComments](#)

Navigates to the comments listing screen and bulk-trashes any comments which exist.

Returns

- **Promise**: Promise resolving once comments have been trashed.

[trashAllPosts](#)

Navigates to the post listing screen and bulk-trashes any posts which exist.

Parameters

- *postType* `string`: – String slug for type of post to trash.
- *postStatus* `string`: – String status of posts to trash.

Returns

- `Promise`: Promise resolving once posts have been trashed.

[uninstallPlugin](#)

Uninstalls a plugin.

Parameters

- *slug* `string`: Plugin slug.

[visitAdminPage](#)

Visits admin page; if user is not logged in then it logs in first, then visits admin page.

Parameters

- *adminPath* `string`: String to be serialized as pathname.
- *query* `string`: String to be serialized as query portion of URL.

[visitSiteEditor](#)

Visits the Site Editor main page

By default, it also skips the welcome guide. The option can be disabled if needed.

Related

- `disableSiteEditorWelcomeGuide`

Parameters

- *query* `string`: String to be serialized as query portion of URL.
- *skipWelcomeGuide* `[boolean]`: Whether to skip the welcome guide as part of the navigation.

[waitForWindowDimensions](#)

Function that waits until the page viewport has the required dimensions. It is being used to address a problem where after using `setViewport` the execution may continue, without the new dimensions being applied. <https://github.com/GoogleChrome/puppeteer/issues/1751>

Parameters

- *width* *number*: Width of the window.
- *height* *number*: Height of the window.

[wpDataSelect](#)

Queries the WordPress data module.

`page.evaluate` – used in the function – returns `undefined` when it encounters a non-serializable value. Since we store many different values in the data module, you can end up with an `undefined` result. Before using this function, make sure the data you are querying doesn't contain non-serializable values, for example, functions, DOM element handles, etc.

Related

- <https://pptr.dev/#?product=Puppeteer&version=v9.0.0&show=api-pageevaluatepagefunction-args>
- <https://github.com/WordPress/gutenberg/pull/31199>

Parameters

- *store* *string*: Store to query e.g: core/editor, core/blocks...
- *selector* *string*: Selector to execute e.g: getBlocks.
- *parameters* ... *Object*: Parameters to pass to the selector.

Returns

- `Promise<?Object>`: Result of querying.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/e2e-test-utils](#)

[Previous: @wordpress/e2e-test-utils-playwright](#) [Previous: @wordpress/e2e-test-utils-playwright](#)
[Next: @wordpress/e2e-tests](#) [Next: @wordpress/e2e-tests](#)

@wordpress/e2e-tests

In this article

Table of Contents

- [Installation](#)
- [Running tests](#)
 - [Run all available tests](#)
 - [Run all available tests and listen for changes.](#)
 - [Run a specific test file](#)
 - [Debugging](#)
- [Contributing to this package](#)

[↑ Back to top](#)

End-To-End (E2E) tests for WordPress.

Note that there's currently an ongoing [project](#) to migrate E2E tests to Playwright instead. This package is deprecated and will only accept bug fixes until fully migrated.

[Installation](#)

Install the module

```
npm install @wordpress/e2e-tests --save-dev
```

[Running tests](#)

The following commands are available on the Gutenberg repo:

```
{  
  "test:e2e": "wp-scripts test-e2e --config packages/e2e-tests/jest.conf  
  "test:e2e:debug": "wp-scripts --inspect-brk test-e2e --config packages/e2e-tests/jest.conf  
  "test:e2e:watch": "npm run test:e2e -- --watch"  
}
```

[Run all available tests](#)

```
npm run test:e2e
```

[Run all available tests and listen for changes.](#)

```
npm run test:e2e:watch
```

Run a specific test file

```
npm run test:e2e -- packages/e2e-test/<path_to_test_file>
# Or, in order to watch for changes:
npm run test:e2e:watch -- packages/e2e-test/<path_to_test_file>
```

Debugging

Makes e2e tests available to debug in a Chrome Browser.

```
npm run test:e2e:debug
```

After running the command, tests will be available for debugging in Chrome by going to chrome://inspect/#devices and clicking `inspect` under the path to `/test-e2e.js`.

Debugging in vscode

Debugging in a Chrome browser can be replaced with vscode's debugger by adding the following configuration to `.vscode/launch.json`:

```
{
  "type": "node",
  "request": "launch",
  "name": "Debug current e2e test",
  "program": "${workspaceFolder}/node_modules/@wordpress/scripts/bin/wp-
  "args": [
    "test-e2e",
    "--config=${workspaceFolder}/packages/e2e-tests/jest.config.js",
    "--verbose=true",
    "--runInBand",
    "--watch",
    "${file}"
  ],
  "console": "integratedTerminal",
  "internalConsoleOptions": "neverOpen",
  "trace": "all"
}
```

This will run jest, targetting the spec file currently open in the editor. vscode's debugger can now be used to add breakpoints and inspect tests as you would in Chrome DevTools.

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/e2e-tests”](#)

[Previous @wordpress/e2e-test-utils](#) [Previous: @wordpress/e2e-test-utils](#)

[Next @wordpress/edit-post](#) [Next: @wordpress/edit-post](#)

@wordpress/edit-post

In this article

[Table of Contents](#)

- [Installation](#)
- [Extending the post editor UI](#)
- [API](#)
 - [initializeEditor](#)
 - [PluginBlockSettingsMenuItem](#)
 - [PluginDocumentSettingPanel](#)
 - [PluginMoreMenuItem](#)
 - [PluginPostPublishPanel](#)
 - [PluginPostStatusInfo](#)
 - [PluginPrePublishPanel](#)
 - [PluginSidebar](#)
 - [PluginSidebarMoreMenuItem](#)
 - [reinitializeEditor](#)
 - [store](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Edit Post Module for WordPress.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

[Installation](#)

Install the module

```
npm install @wordpress/edit-post
```

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.

Extending the post editor UI

Extending the editor UI can be accomplished with the `registerPlugin` API, allowing you to define all your plugin's UI elements in one place.

Refer to [the plugins module documentation](#) for more information.

The components exported through the API can be used with the `registerPlugin` ([see documentation](#)) API.

They can be found in the global variable `wp.editPost` when defining `wp-edit-post` as a script dependency.

API

initializeEditor

Initializes and returns an instance of Editor.

Parameters

- `id` `string`: Unique identifier for editor instance.
- `postType` `string`: Post type of the post to edit.
- `postId` `Object`: ID of the post to edit.
- `settings` `?Object`: Editor settings object.
- `initialEdits` `Object`: Programmatic edits to apply initially, to be considered as non-user-initiated (bypass for unsaved changes prompt).

PluginBlockSettingsMenuItem

Renders a new item in the block settings menu.

Usage

```
// Using ES5 syntax
var __ = wp.i18n.__;
var PluginBlockSettingsMenuItem = wp.editPost.PluginBlockSettingsMenuItem;

function doOnClick() {
    // To be called when the user clicks the menu item.
}

function MyPluginBlockSettingsMenuItem() {
    return React.createElement( PluginBlockSettingsMenuItem, {
        allowedBlocks: [ 'core/paragraph' ],
        icon: 'dashicon-name',
        label: __( 'Menu item text' ),
        onClick: doOnClick,
    } );
}
```

```
// Using ESNext syntax
import { __ } from '@wordpress/i18n';
import { PluginBlockSettingsMenuItem } from '@wordpress/edit-post';

const doOnClick = () => {
    // To be called when the user clicks the menu item.
};

const MyPluginBlockSettingsMenuItem = () => (
    <PluginBlockSettingsMenuItem
        allowedBlocks={[ 'core/paragraph' ] }
        icon="dashicon-name"
        label={ __( 'Menu item text' ) }
        onClick={ doOnClick }
    />
);

```

Parameters

- *props Object*: Component props.
- *props.allowedBlocks [Array]*: An array containing a list of block names for which the item should be shown. If not present, it'll be rendered for any block. If multiple blocks are selected, it'll be shown if and only if all of them are in the allowed list.
- *props.icon [WPBlockTypeIconRender]*: The [Dashicon](#) icon slug string, or an SVG WP element.
- *props.label string*: The menu item text.
- *props.onClick Function*: Callback function to be executed when the user click the menu item.
- *props.small [boolean]*: Whether to render the label or not.
- *props.role [string]*: The ARIA role for the menu item.

Returns

- Component: The component to be rendered.

[**PluginDocumentSettingPanel**](#)

Renders items below the Status & Availability panel in the Document Sidebar.

Usage

```
// Using ES5 syntax
var el = React.createElement;
var __ = wp.i18n.__;
var registerPlugin = wp.plugins.registerPlugin;
var PluginDocumentSettingPanel = wp.editPost.PluginDocumentSettingPanel;

function MyDocumentSettingPlugin() {
    return el(
        PluginDocumentSettingPanel,
        {
            className: 'my-document-setting-plugin',
            title: 'My Panel',
            name: 'my-panel',
        }
    );
}
```

```

        },
        __( 'My Document Setting Panel' )
    );
}

registerPlugin( 'my-document-setting-plugin', {
    render: MyDocumentSettingPlugin,
} );

// Using ESNext syntax
import { registerPlugin } from '@wordpress/plugins';
import { PluginDocumentSettingPanel } from '@wordpress/edit-post';

const MyDocumentSettingTest = () => (
    <PluginDocumentSettingPanel
        className="my-document-setting-plugin"
        title="My Panel"
        name="my-panel"
    >
        <p>My Document Setting Panel</p>
    </PluginDocumentSettingPanel>
);
registerPlugin( 'document-setting-test', { render: MyDocumentSettingTest } );

```

Parameters

- *props Object*: Component properties.
- *props.name string*: Required. A machine-friendly name for the panel.
- *props.className [string]*: An optional class name added to the row.
- *props.title [string]*: The title of the panel
- *props.icon [WPBlockTypeIconRender]*: The [Dashicon](#) icon slug string, or an SVG WP element, to be rendered when the sidebar is pinned to toolbar.
- *props.children Element*: Children to be rendered

Returns

- *Component*: The component to be rendered.

[PluginMoreMenuItem](#)

Renders a menu item in `Plugins` group in `More` Menu drop down, and can be used to as a button or link depending on the props provided. The text within the component appears as the menu item label.

Usage

```

// Using ES5 syntax
var __ = wp.i18n.__;
var PluginMoreMenuItem = wp.editPost.PluginMoreMenuItem;
var moreIcon = React.createElement( 'svg' ); //... svg element.

function onButtonClick() {
    alert( 'Button clicked.' );
}

```

```

}

function MyButtonMoreMenuItem() {
    return React.createElement(
        PluginMoreMenuItem,
        {
            icon: moreIcon,
            onClick: onButtonClick,
        },
        __('My button title')
    );
}

// Using ESNext syntax
import { __ } from '@wordpress/i18n';
import { PluginMoreMenuItem } from '@wordpress/edit-post';
import { more } from '@wordpress/icons';

function onButtonClick() {
    alert('Button clicked.');
}

const MyButtonMoreMenuItem = () => (
    <PluginMoreMenuItem icon={ more } onClick={ onButtonClick }>
        { __('My button title') }
    </PluginMoreMenuItem>
);

```

Parameters

- *props Object*: Component properties.
- *props.href [string]*: When `href` is provided then the menu item is represented as an anchor rather than button. It corresponds to the `href` attribute of the anchor.
- *props.icon [WPBlockTypeIconRender]*: The [Dashicon](#) icon slug string, or an SVG WP element, to be rendered to the left of the menu item label.
- *props.onClick [Function]*: The callback function to be executed when the user clicks the menu item.
- *props.other [...]*: Any additional props are passed through to the underlying [MenuItem](#) component.

Returns

- **Component**: The component to be rendered.

[**PluginPostPublishPanel**](#)

Renders provided content to the post-publish panel in the publish flow (side panel that opens after a user publishes the post).

Usage

```
// Using ES5 syntax
var __ = wp.i18n.__;
var PluginPostPublishPanel = wp.editPost.PluginPostPublishPanel;
```

```

function MyPluginPostPublishPanel() {
    return React.createElement(
        PluginPostPublishPanel,
        {
            className: 'my-plugin-post-publish-panel',
            title: __( 'My panel title' ),
            initialOpen: true,
        },
        __( 'My panel content' )
    );
}

// Using ESNext syntax
import { __ } from '@wordpress/i18n';
import { PluginPostPublishPanel } from '@wordpress/edit-post';

const MyPluginPostPublishPanel = () => (
    <PluginPostPublishPanel
        className="my-plugin-post-publish-panel"
        title={ __( 'My panel title' ) }
        initialOpen={ true }
    >
        { __( 'My panel content' ) }
    </PluginPostPublishPanel>
);

```

Parameters

- *props Object*: Component properties.
- *props.className [string]*: An optional class name added to the panel.
- *props.title [string]*: Title displayed at the top of the panel.
- *props.initialOpen [boolean]*: Whether to have the panel initially opened. When no title is provided it is always opened.
- *props.icon [WPBlockTypeIconRender]*: The [Dashicon](#) icon slug string, or an SVG WP element, to be rendered when the sidebar is pinned to toolbar.
- *props.children Element*: Children to be rendered

Returns

- *Component*: The component to be rendered.

[PluginPostStatusInfo](#)

Renders a row in the Summary panel of the Document sidebar. It should be noted that this is named and implemented around the function it serves and not its location, which may change in future iterations.

Usage

```
// Using ES5 syntax
var __ = wp.i18n.__;
var PluginPostStatusInfo = wp.editPost.PluginPostStatusInfo;
```

```

function MyPluginPostStatusInfo() {
    return React.createElement(
        PluginPostStatusInfo,
        {
            className: 'my-plugin-post-status-info',
        },
        __('My post status info')
    );
}

// Using ESNext syntax
import { __ } from '@wordpress/i18n';
import { PluginPostStatusInfo } from '@wordpress/edit-post';

const MyPluginPostStatusInfo = () => (
    <PluginPostStatusInfo className="my-plugin-post-status-info">
        { __( 'My post status info' ) }
    </PluginPostStatusInfo>
);

```

Parameters

- *props Object*: Component properties.
- *props.className [string]*: An optional class name added to the row.
- *props.children Element*: Children to be rendered.

Returns

- Component: The component to be rendered.

[PluginPrePublishPanel](#)

Renders provided content to the pre-publish side panel in the publish flow (side panel that opens when a user first pushes “Publish” from the main editor).

Usage

```

// Using ES5 syntax
var __ = wp.i18n.__;
var PluginPrePublishPanel = wp.editPost.PluginPrePublishPanel;

function MyPluginPrePublishPanel() {
    return React.createElement(
        PluginPrePublishPanel,
        {
            className: 'my-plugin-pre-publish-panel',
            title: __( 'My panel title' ),
            initialOpen: true,
        },
        __( 'My panel content' )
    );
}

```

```
// Using ESNext syntax
import { __ } from '@wordpress/i18n';
import { PluginPrePublishPanel } from '@wordpress/edit-post';

const MyPluginPrePublishPanel = () => (
  <PluginPrePublishPanel
    className="my-plugin-pre-publish-panel"
    title={ __( 'My panel title' ) }
    initialOpen={ true }
  >
  { __( 'My panel content' ) }
</PluginPrePublishPanel>
);


```

Parameters

- *props Object*: Component props.
- *props.className [string]*: An optional class name added to the panel.
- *props.title [string]*: Title displayed at the top of the panel.
- *props.initialOpen [boolean]*: Whether to have the panel initially opened. When no title is provided it is always opened.
- *props.icon [WPBlockTypeIconRender]*: The [Dashicon](#) icon slug string, or an SVG WP element, to be rendered when the sidebar is pinned to toolbar.
- *props.children Element*: Children to be rendered

Returns

- Component: The component to be rendered.

[PluginSidebar](#)

Renders a sidebar when activated. The contents within the `PluginSidebar` will appear as content within the sidebar. It also automatically renders a corresponding `PluginSidebarMenuItem` component when `isPinnable` flag is set to `true`. If you wish to display the sidebar, you can with use the `PluginSidebarMoreMenuItem` component or the `wp.data.dispatch` API:

```
wp.data
  .dispatch( 'core/edit-post' )
  .openGeneralSidebar( 'plugin-name/sidebar-name' );
```

Related

- `PluginSidebarMoreMenuItem`

Usage

```
// Using ES5 syntax
var __ = wp.i18n.__;
var el = React.createElement;
var PanelBody = wp.components.PanelBody;
var PluginSidebar = wp.editPost.PluginSidebar;
var moreIcon = React.createElement( 'svg' ); //... svg element.
```

```

function MyPluginSidebar() {
    return el(
        PluginSidebar,
        {
            name: 'my-sidebar',
            title: 'My sidebar title',
            icon: moreIcon,
        },
        el( PanelBody, {}, __( 'My sidebar content' ) )
    );
}

// Using ESNext syntax
import { __ } from '@wordpress/i18n';
import { PanelBody } from '@wordpress/components';
import { PluginSidebar } from '@wordpress/edit-post';
import { more } from '@wordpress/icons';

const MyPluginSidebar = () => (
    <PluginSidebar name="my-sidebar" title="My sidebar title" icon={ more }
        <PanelBody>{ __( 'My sidebar content' ) }</PanelBody>
    </PluginSidebar>
);

```

Parameters

- *props Object*: Element props.
- *props.name string*: A string identifying the sidebar. Must be unique for every sidebar registered within the scope of your plugin.
- *props.className [string]*: An optional class name added to the sidebar body.
- *props.title string*: Title displayed at the top of the sidebar.
- *props.isPinnable [boolean]*: Whether to allow to pin sidebar to the toolbar. When set to true it also automatically renders a corresponding menu item.
- *props.icon [WPBlockTypeIconRender]*: The [Dashicon](#) icon slug string, or an SVG WP element, to be rendered when the sidebar is pinned to toolbar.

[PluginSidebarMoreMenuItem](#)

Renders a menu item in Plugins group in More Menu drop down, and can be used to activate the corresponding `PluginSidebar` component. The text within the component appears as the menu item label.

Usage

```

// Using ES5 syntax
var __ = wp.i18n.__;
var PluginSidebarMoreMenuItem = wp.editPost.PluginSidebarMoreMenuItem;
var moreIcon = React.createElement( 'svg' ); //... svg element.

function MySidebarMoreMenuItem() {
    return React.createElement(
        PluginSidebarMoreMenuItem,
        {
            target: 'my-sidebar',

```

```

        icon: moreIcon,
    },
    __( 'My sidebar title' )
);
}

// Using ESNext syntax
import { __ } from '@wordpress/i18n';
import { PluginSidebarMoreMenuItem } from '@wordpress/edit-post';
import { more } from '@wordpress/icons';

const MySidebarMoreMenuItem = () => (
    <PluginSidebarMoreMenuItem target="my-sidebar" icon={ more }>
        { __( 'My sidebar title' ) }
    </PluginSidebarMoreMenuItem>
);

```

Parameters

- *props Object*: Component props.
- *props.target string*: A string identifying the target sidebar you wish to be activated by this menu item. Must be the same as the `name` prop you have given to that sidebar.
- *props.icon [WPBlockTypeIconRender]*: The [Dashicon](#) icon slug string, or an SVG WP element, to be rendered to the left of the menu item label.

Returns

- Component: The component to be rendered.

[reinitializeEditor](#)

Used to reinitialize the editor after an error. Now it's a deprecated noop function.

[store](#)

Store definition for the edit post namespace.

Related

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

Type

- Object

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/edit-post](#)

[Previous](#) [@wordpress/e2e-tests](#) [Previous: @wordpress/e2e-tests](#)
[Next](#) [@wordpress/edit-site](#) [Next: @wordpress/edit-site](#)

@wordpress/edit-site

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Edit Site Page Module for WordPress.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

[Installation](#)

`npm install @wordpress/edit-site`

[Usage](#)

```
/**  
 * WordPress dependencies  
 */  
import { initialize } from '@wordpress/edit-site';  
  
/**  
 * Internal dependencies  
 */  
import blockEditorSettings from './block-editor-settings';
```

```
initialize( '#editor-root', blockEditorSettings );
```

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.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/edit-site](#)

[Previous @wordpress/edit-post](#) [Previous: @wordpress/edit-post](#)
[Next @wordpress/edit-widgets](#) [Next: @wordpress/edit-widgets](#)

[@wordpress/edit-widgets](#)

In this article

Table of Contents

- [Batch processing](#)
- [Installation](#)
- [How this works](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Widgets Page Module for WordPress.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

Batch processing

This package contains the first version of what may eventually become @wordpress/batch-processing package. Once imported, core/_experimental-batch-processing store gets registered. As the name says – it is highly experimental and considered a private API for now.

Installation

Install the module

```
npm install @wordpress/edit-widgets
```

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.

How this works

The new Widgets screen in WordPress admin is another block editor, just like the Post editor or the experimental site editor. Hence it will be referred often as the Widgets editor.

This editor manages widget areas and offers a way to add Gutenberg blocks to them, in addition to regular widgets. To support both widgets and blocks, the editor employs a translation mechanism between widget storage and block grammar.

There is a widget block that acts as a block UI for the widget data. This block is instantiated by default with a list of all available widgets to choose from. The block wraps its functionality in two modes: edit and preview based on the selected widget. The widget block's edit mode shows the standard Widget form, while the preview does a server-side render of the widget.

There is a block widget that acts as a storage mechanism for blocks added to widget areas. This widget is a special case of the HTML widget, where the block data is stored as it is rendered by the block's save function. All blocks added to widget areas are stored as these special HTML widgets, in one type of widget, the block widget.

This mechanism, using a widget block to edit widgets as blocks and a block widget to store blocks as widgets, ensures 100% compatibility with the old Widgets screen. Thus, if the new Widget editor, which is block-based, breaks some widgets' functionality that depends on the admin page's HTML structure or jQuery events, it is easy to revert to the old screen and continue to edit the legacy widgets.

Being just a block editor, the Widgets editor needs REST API entity management endpoints. For support, two new endpoints have been added: ./widgets and /sidebars. The ./widgets endpoint is used to load and save widgets and retrieve a server-side render of the widget's edit form. The /sidebars endpoint is used to list widget areas and assign or remove a widget to or from a widget area. There is also an /widget-types endpoint listing what kind widgets are available, e.g. text widget, calendar widget etc

In order to make the experience as seamless as possible for users, the following “magic” happens in the Widgets editor:

- for every available widget, a variation of the widget block is registered so that the user can see and search by the exact name of what they need
- all widgets that have a block equivalent (a block that fulfills the same function) can be made not available as a widget block variation via a filter
- all core widgets that have a block equivalent are not available as a widget block variation

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/edit-widgets](#)”

[Previous @wordpress/edit-site](#) [Previous: @wordpress/edit-site](#)
[Next @wordpress/editor](#) [Next: @wordpress/editor](#)

@wordpress/editor

In this article

Table of Contents

- [Installation](#)
- [How it works](#)
- [Components](#)
 - [BlockControls](#)
 - [RichText](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This module utilizes components from the `@wordpress/block-editor` package. Having an awareness of the concept of a WordPress post, it associates the loading and saving mechanism of

the value representing blocks to a post and its content. It also provides various components relevant for working with a post object in the context of an editor (e.g., a post title input component). This package can support editing posts of any post type and does not assume that rendering happens in any particular WordPress screen or layout arrangement.

Installation

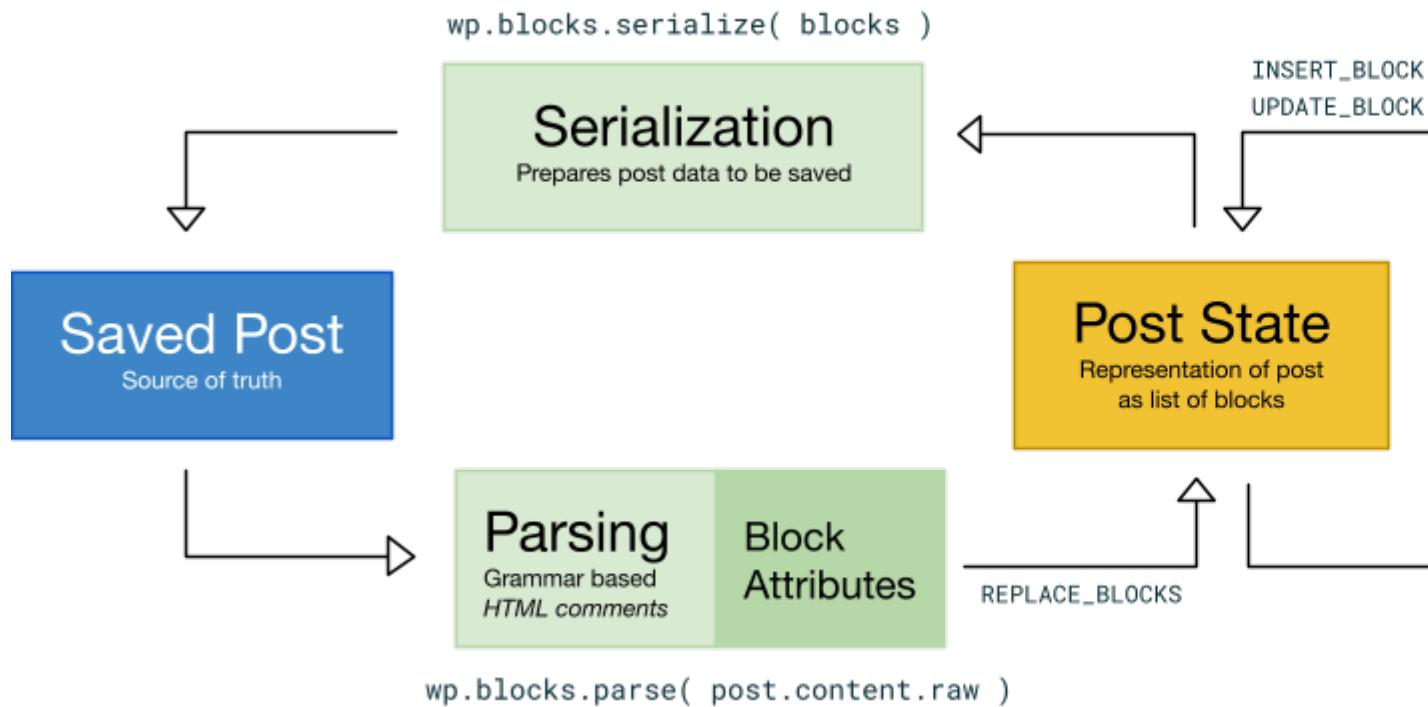
Install the module

```
npm install @wordpress/editor --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.*

How it works

The logic flow concerning the editor includes: inferring a block representation of the post content (parsing); describing the state of a post (representation); rendering of the post to the DOM (rendering); attaching controls to manipulate the content a.k.a blocks (UI).



The goal of the editor element is to let the user manipulate the content of their posts in a deterministic way—organized through the presence of blocks of content. Usually, in a declarative flow, the pieces that compose a post would be represented in a certain order and the machine would be able to generate an output view from it with the necessary UI controls. However, we don't begin in WordPress with a representation of the state of the post that is conducive to this

expression nor one that even has any knowledge of blocks because content is stored in a serialized way in a single field.

Such a crucial step is handled by the grammar parsing which takes the serialized content of the post and infers an ordered block list using, preferably, syntax hints present in HTML comments. The editor is initialized with a state representation of the block nodes generated by the parsing of the raw content of a post element: `wp.blocks.parse(post.content.raw)`.

The *visual editor* is thus a component that contains and renders the list of block nodes from the internal state into the page. This removes any trace of imperative handling when it comes to finding a block and manipulating a block. As a matter of fact, the visual editor or the text editor are just two different—equally valid—views of the same representation of state. The internal representation of the post content is updated as blocks are updated and it is serialized back to be saved in `post_content`.

Individual blocks are handled by the `VisualBlock` component, which attaches event handlers and renders the `edit` function of a block definition to the document with the corresponding attributes and local state. The `edit` function is the markup shape of a component while in editing mode.

Components

Because many blocks share the same complex behaviors, reusable components are made available to simplify implementations of your block's `edit` function.

BlockControls

When returned by your block's `edit` implementation, renders a toolbar of icon buttons. This is useful for block-level modifications to be made available when a block is selected. For example, if your block supports alignment, you may want to display alignment options in the selected block's toolbar.

Example:

```
( function ( editor, React ) {
    var el = React.createElement,
        BlockControls = editor.BlockControls,
        AlignmentToolbar = editor.AlignmentToolbar;

    function edit( props ) {
        return [
            // Controls: (only visible when block is selected)
            el(
                BlockControls,
                { key: 'controls' },
                el( AlignmentToolbar, {
                    value: props.align,
                    onChange: function ( nextAlign ) {
                        props.setAttributes( { align: nextAlign } );
                    },
                } )
            ),
        ],
    }
})
```

```

        // Block content: (with alignment as attribute)
      el(
        'p',
        { key: 'text', style: { textAlign: props.align } },
        'Hello World!'
      ),
    ];
}
})( window.wp.editor, window.React );

```

Note in this example that we render `AlignmentToolbar` as a child of the `BlockControls` element. This is another pre-configured component you can use to simplify block text alignment.

Alternatively, you can create your own toolbar controls by passing an array of `controls` as a prop to the `BlockControls` component. Each control should be an object with the following properties:

- `icon: string` – Slug of the Dashicon to be shown in the control's toolbar button
- `title: string` – A human-readable localized text to be shown as the tooltip label of the control's button
- `subscript: ?string` – Optional text to be shown adjacent the button icon as subscript (for example, heading levels)
- `isActive: ?boolean` – Whether the control should be considered active / selected. Defaults to `false`.

To create divisions between sets of controls within the same `BlockControls` element, passing `controls` instead as a nested array (array of arrays of objects). A divider will be shown between each set of controls.

[RichText](#)

Render a rich [contenteditable input](#), providing users the option to add emphasis to content or links to content. It behaves similarly to a [controlled component](#), except that `onChange` is triggered less frequently than would be expected from a traditional `input` field, usually when the user exits the field.

The following properties (non-exhaustive list) are made available:

- `value: string` – Markup value of the field. Only valid markup is allowed, as determined by `inline` value and available controls.
- `onChange: Function` – Callback handler when the value of the field changes, passing the new value as its only argument.
- `placeholder: string` – A text hint to be shown to the user when the field value is empty, similar to the [input and textarea attribute of the same name](#).

Example:

```

( function ( editor, React ) {
  var el = React.createElement,
      RichText = editor.RichText;

  function edit( props ) {
    function onChange( value ) {

```

```
        props.setAttributes( { text: value } );
    }

    return el( RichText, {
        value: props.attributes.text,
        onChange: onChange,
    } );
}

// blocks.registerBlockType( ..., { edit: edit, ... } );
} )( window.wp.editor, window.React );
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/editor](#)

[Previous](#) [@wordpress/edit-widgets](#) [Previous: @wordpress/edit-widgets](#)

[Next](#) [@wordpress/experiments](#) [Next: @wordpress/experiments](#)

@wordpress/experiments

In this article

[Table of Contents](#)

- [Getting started](#)
- [Shipping experimental APIs](#)
- [Technical limitations](#)
- [Contributing to this package](#)

[↑ Back to top](#)

[@wordpress/experiments](#) enables sharing private `__experimental` APIs across @wordpress packages without publicly exposing them to WordPress extenders.

Getting started

Every `@wordpress` package wanting to privately access or expose experimental APIs must opt-in to `@wordpress/experiments`:

```
// In packages/block-editor/experiments.js:  
import { __dangerousOptInToUnstableAPIsOnlyForCoreModules } from '@wordpress/experiments'  
export const { lock, unlock } =  
  __dangerousOptInToUnstableAPIsOnlyForCoreModules(  
    'I know using unstable features means my plugin or theme will inevitably break on the next WordPress release.',  
    '@wordpress/block-editor' // Name of the package calling __dangerousOptInToUnstableAPIsOnlyForCoreModules  
    // (not the name of the package whose API is being exposed)  
  );
```

Each package may only opt in once. The function name communicates that plugins are not supposed to use it.

The function will throw an error if the following conditions are not met:

1. The first argument must exactly match the required consent string: 'I know using unstable features means my plugin or theme will inevitably break on the next WordPress release.'
2. The second argument must be a known `@wordpress` package that hasn't yet opted into `@wordpress/experiments`

Once the opt-in is complete, the obtained `lock()` and `unlock()` utilities enable hiding `__experimental` APIs from the naked eye:

```
// Say this object is exported from a package:  
export const publicObject = {};  
  
// However, this string is internal and should not be publicly available:  
const __experimentalString = '__experimental information';  
  
// Solution: lock the string "inside" of the object:  
lock( publicObject, __experimentalString );  
  
// 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 ) );  
// "__experimental information"  
  
// lock() accepts all data types, not just strings:  
export const anotherObject = {};  
lock( anotherObject, function __experimentalFn() {} );  
console.log( unlock( anotherObject ) );  
// function __experimentalFn() {}
```

Use `lock()` and `unlock()` to privately distribute the `__experimental` APIs across `@wordpress` packages:

```
// In packages/package1/index.js:  
import { lock } from './experiments';  
  
export const experiments = {};  
/* Attach private data to the exported object */  
lock(experiments, {  
    __experimentalFunction: function() {},  
});  
  
// In packages/package2/index.js:  
import { experiments } from '@wordpress/package1';  
import { unlock } from './experiments';  
  
const {  
    __experimentalFunction  
} = unlock( experiments );
```

[Shipping experimental APIs](#)

See the [Experimental and Unstable APIs chapter of Coding Guidelines](#) to learn how `lock()` and `unlock()` can help you ship private experimental functions, arguments, components, properties, actions, selectors.

[Technical limitations](#)

A determined developer who would want to use the private experimental APIs at all costs would have to:

- Realize a private importing system exists
- Read the code where the risks would be spelled out in capital letters
- Explicitly type out he or she is aware of the consequences
- Pretend to register a `@wordpress` package (and trigger an error as soon as the real package is loaded)

Dangerously opting in to using these APIs by theme and plugin developers is not recommended. Furthermore, the WordPress Core philosophy to strive to maintain backward compatibility for third-party developers **does not apply** to experimental APIs registered via this package.

The consent string for opting in to these APIs may change at any time and without notice. This change will break existing third-party code. Such a change may occur in either a major or minor release.

[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](#).

First published

September 26, 2022

Last updated

February 9, 2023

Edit article

[Improve it on GitHub: @wordpress/experiments”](#)

[Previous @wordpress/editor](#) [Previous: @wordpress/editor](#)

[Next @wordpress/element](#) [Next: @wordpress/element](#)

@wordpress/element

In this article

[Table of Contents](#)

- [Installation](#)
- [Why React?](#)
- [API](#)
 - [Children](#)
 - [cloneElement](#)
 - [Component](#)
 - [concatChildren](#)
 - [createContext](#)
 - [createElement](#)
 - [createInterpolateElement](#)
 - [createPortal](#)
 - [createRef](#)
 - [createRoot](#)
 - [findDOMNode](#)
 - [flushSync](#)
 - [forwardRef](#)
 - [Fragment](#)
 - [hydrate](#)
 - [hydrateRoot](#)
 - [isEmptyElement](#)
 - [isValidElement](#)
 - [lazy](#)
 - [memo](#)
 - [Platform](#)
 - [PureComponent](#)
 - [RawHTML](#)
 - [render](#)
 - [renderToString](#)
 - [startTransition](#)

- [StrictMode](#)
- [Suspense](#)
- [switchChildrenNodeName](#)
- [unmountComponentAtNode](#)
- [useCallback](#)
- [useContext](#)
- [useDebugValue](#)
- [useDeferredValue](#)
- [useEffect](#)
- [useId](#)
- [useImperativeHandle](#)
- [useInsertionEffect](#)
- [useLayoutEffect](#)
- [useMemo](#)
- [useReducer](#)
- [useRef](#)
- [useState](#)
- [useSyncExternalStore](#)
- [useTransition](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Element is a package that builds on top of [React](#) and provide a set of utilities to work with React components and React elements.

[**Installation**](#)

Install the module

```
npm install @wordpress/element --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.*

[**Why React?**](#)

At the risk of igniting debate surrounding any single “best” front-end framework, the choice to use any tool should be motivated specifically to serve the requirements of the system. In modeling the concept of a [block](#), we observe the following technical requirements:

- An understanding of a block in terms of its underlying values (in the [random image example](#), a category)
- A means to describe the UI of a block given these values

At its most basic, React provides a simple input / output mechanism. **Given a set of inputs (“props”), a developer describes the output to be shown on the page.** This is most elegantly observed in its [function components](#). React serves the role of reconciling the desired output with the current state of the page.

The offerings of any framework necessarily become more complex as these requirements increase; many front-end frameworks prescribe ideas around page routing, retrieving and updating data, and managing layout. React is not immune to this, but the introduced complexity is rarely caused by React itself, but instead managing an arrangement of supporting tools. By moving these concerns out of sight to the internals of the system (WordPress core code), we can minimize the responsibilities of plugin authors to a small, clear set of touch points.

[API](#)

[Children](#)

Object that provides utilities for dealing with React children.

[cloneElement](#)

Creates a copy of an element with extended props.

Parameters

- *element* Element: Element
- *props* ?Object: Props to apply to cloned element

Returns

- Element: Cloned element.

[Component](#)

A base class to create WordPress Components (Refs, state and lifecycle hooks)

[concatChildren](#)

Concatenate two or more React children objects.

Parameters

- *childrenArguments* . . . ?Object: Array of children arguments (array of arrays/strings/objects) to concatenate.

Returns

- Array: The concatenated value.

[createContext](#)

Creates a context object containing two components: a provider and consumer.

Parameters

- *defaultValue* Object: A default data stored in the context.

Returns

- `Object`: Context object.

createElement

Returns a new element of given type. Type can be either a string tag name or another function which itself returns an element.

Parameters

- `type ? (string | Function)`: Tag name or element creator
- `props Object`: Element properties, either attribute set to apply to DOM node or values to pass through to element creator
- `children ... Element`: Descendant elements

Returns

- `Element`: Element.

createInterpolateElement

This function creates an interpolated element from a passed in string with specific tags matching how the string should be converted to an element via the conversion map value.

Usage

For example, for the given string:

“This is a string with a link and a self-closing tag”

You would have something like this as the conversionMap value:

```
{  
  span: <span />,  
  a: <a href={ 'https://github.com' } />,  
  CustomComponentB: <CustomComponent />,  
}
```

Parameters

- `interpolatedString string`: The interpolation string to be parsed.
- `conversionMap Record<string, Element>`: The map used to convert the string to a react element.

Returns

- `Element`: A wp element.

createPortal

Creates a portal into which a component can be rendered.

Related

- <https://github.com/facebook/react/issues/10309#issuecomment-318433235>

Parameters

- *child import('react').ReactElement*: Any renderable child, such as an element, string, or fragment.
- *container HTMLElement*: DOM node into which element should be rendered.

[createRef](#)

Returns an object tracking a reference to a rendered element via its `current` property as either a `DOMElement` or `Element`, dependent upon the type of element rendered with the `ref` attribute.

Returns

- `Object`: Ref object.

[createRoot](#)

Creates a new React root for the target DOM node.

Related

- <https://react.dev/reference/react-dom/client/createRoot>

Changelog

6.2.0 Introduced in WordPress core.

[findDOMNode](#)

Finds the dom node of a React component.

Parameters

- *component import('react').ComponentType*: Component's instance.

[flushSync](#)

Forces React to flush any updates inside the provided callback synchronously.

Parameters

- *callback Function*: Callback to run synchronously.

[forwardRef](#)

Component enhancer used to enable passing a ref to its wrapped component. Pass a function argument which receives `props` and `ref` as its arguments, returning an element using the forwarded ref. The return value is a new component which forwards its ref.

Parameters

- **forwarder Function:** Function passed `props` and `ref`, expected to return an element.

Returns

- **Component:** Enhanced component.

Fragment

A component which renders its children without any wrapping element.

hydrate

Deprecated since WordPress 6.2.0. Use `hydrateRoot` instead.

Hydrates a given element into the target DOM node.

Related

- <https://react.dev/reference/react-dom/hydrate>

hydrateRoot

Creates a new React root for the target DOM node and hydrates it with a pre-generated markup.

Related

- <https://react.dev/reference/react-dom/client/hydrateRoot>

Changelog

6.2.0 Introduced in WordPress core.

isEmptyElement

Checks if the provided WP element is empty.

Parameters

- `element` *: WP element to check.

Returns

- `boolean`: True when an element is considered empty.

isValidElement

Checks if an object is a valid React Element.

Parameters

- `objectToCheck Object`: The object to be checked.

Returns

- boolean: true if objectToTest is a valid React Element and false otherwise.

lazy

Related

- <https://reactjs.org/docs/react-api.html#reactlazy>

memo

Related

- <https://reactjs.org/docs/react-api.html#reactmemo>

Platform

Component used to detect the current Platform being used. Use Platform.OS === ‘web’ to detect if running on web environment.

This is the same concept as the React Native implementation.

Related

- <https://facebook.github.io/react-native/docs/platform-specific-code#platform-module> Here is an example of how to use the select method:

Usage

```
import { Platform } from '@wordpress/element';

const placeholderLabel = Platform.select( {
    native: __( 'Add media' ),
    web: __(
        'Drag images, upload new ones or select files from your library.'
    ),
} );
```

PureComponent

Related

- <https://reactjs.org/docs/react-api.html#reactpurecomponent>

RawHTML

Component used as equivalent of Fragment with unescaped HTML, in cases where it is desirable to render dangerous HTML without needing a wrapper element. To preserve additional props, a `div` wrapper *will* be created if any props aside from `children` are passed.

Parameters

- `props RawHTMLProps`: Children should be a string of HTML or an array of strings. Other props will be passed through to the div wrapper.

Returns

- `JSX.Element`: Dangerously-rendering component.

[render](#)

Deprecated since WordPress 6.2.0. Use `createRoot` instead.

Renders a given element into the target DOM node.

Related

- <https://react.dev/reference/react-dom/render>

[renderToString](#)

Serializes a React element to string.

Parameters

- `element import('react').ReactNode`: Element to serialize.
- `context [Object]`: Context object.
- `legacyContext [Object]`: Legacy context object.

Returns

- `string`: Serialized element.

[startTransition](#)

Related

- <https://reactjs.org/docs/react-api.html#starttransition>

[StrictMode](#)

Component that activates additional checks and warnings for its descendants.

[Suspense](#)

Related

- <https://reactjs.org/docs/react-api.html#reactsuspense>

[switchChildrenNodeName](#)

Switches the nodeName of all the elements in the children object.

Parameters

- *children* ?Object: Children object.
- *nodeName* string: Node name.

Returns

- ?Object: The updated children object.

[unmountComponentAtNode](#)

Deprecated since WordPress 6.2.0. Use `root.unmount()` instead.

Removes any mounted element from the target DOM node.

Related

- <https://react.dev/reference/react-dom/unmountComponentAtNode>

[useCallback](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usecallback>

[useContext](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usecontext>

[useDebugValue](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usedebugvalue>

[useDeferredValue](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usedeferredvalue>

[useEffect](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#useeffect>

[useId](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#useid>

[useImperativeHandle](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#useimperativehandle>

[useInsertionEffect](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#useinsertioneffect>

[useLayoutEffect](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#uselayouтеffект>

[useMemo](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usememo>

[useReducer](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usereducer>

[useRef](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#useref>

[useState](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usestate>

[useSyncExternalStore](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usesyncexternalstore>

[useTransition](#)

Related

- <https://reactjs.org/docs/hooks-reference.html#usetransition>

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/element”](#)

[Previous @wordpress/experiments](#) [Previous: @wordpress/experiments](#)
[Next @wordpress/env](#) [Next: @wordpress/env](#)

@wordpress/env

In this article

Table of Contents

- [Quick \(tl;dr\) instructions](#)
- [Prerequisites](#)
- [Installation](#)
 - [Installation as a global package](#)
 - [Installation as a local package](#)
- [Usage](#)
 - [Starting the environment](#)
 - [Stopping the environment](#)
- [Troubleshooting common problems](#)
 - [1. Check that wp-env is running](#)
 - [2. Check the port number](#)
 - [3. Restart wp-env with updates](#)
 - [4. Restart Docker](#)
 - [5. Reset the database](#)
 - [6. Destroy everything and start again 🔥](#)
- [Using included WordPress PHPUnit test files](#)
 - [Customizing the wp-tests-config.php file](#)
- [Using composer, phpunit, and wp-cli tools.](#)
- [Using Xdebug](#)
 - [Xdebug IDE support](#)

- [Command reference](#)
 - [wp-env start](#)
 - [wp-env stop](#)
 - [wp-env clean \[environment\]](#)
 - [wp-env run <container> \[command...\]](#)
 - [wp-env destroy](#)
 - [wp-env logs \[environment\]](#)
 - [wp-env install-path](#)
- [.wp-env.json](#)
- [.wp-env.override.json](#)
- [Default wp-config values.](#)
- [Lifecycle Scripts](#)
- [Examples](#)
 - [Latest stable WordPress + current directory as a plugin](#)
 - [Latest development WordPress + current directory as a plugin](#)
 - [Local wordpress-develop + current directory as a plugin](#)
 - [A complete testing environment](#)
 - [Add mu-plugins and other mapped directories](#)
 - [Avoid activating plugins or themes on the instance](#)
 - [Map a plugin only in the tests environment](#)
 - [Custom Port Numbers](#)
 - [Specific PHP Version](#)
 - [Node Lifecycle Script](#)
 - [Advanced PHP settings](#)
- [Contributing to this package](#)

[↑ Back to top](#)

wp-env lets you easily set up a local WordPress environment for building and testing plugins and themes. It's simple to install and requires no configuration.

[Quick \(tl;dr\) instructions](#)

Ensure that Docker is running, then:

```
$ cd /path/to/a/wordpress/plugin
$ npm -g i @wordpress/env
$ wp-env start
```

The local environment will be available at <http://localhost:8888> (Username: `admin`, Password: `password`).

The database credentials are: user `root`, password `password`. For a comprehensive guide on connecting directly to the database, refer to [Accessing the MySQL Database](#).

[Prerequisites](#)

wp-env relies on a few commonly used developer tools:

- **Docker.** wp-env is powered by Docker. There are instructions available for installing Docker on [Windows](#) (we recommend the WSL2 backend), [macOS](#), and [Linux](#).

- **Node.js.** wp-env is written as a Node script. We recommend using a Node version manager like [nvm](#) to install the latest LTS version. Alternatively, you can [download it directly here](#).
- **git.** Git is used for downloading software from source control, such as WordPress, plugins, and themes. [You can find the installation instructions here.](#)

Installation

Installation as a global package

After confirming that the prerequisites are installed, you can install wp-env globally like so:

```
$ npm -g i @wordpress/env
```

You're now ready to use wp-env!

Installation as a local package

If your project already has a package.json, it's also possible to use wp-env as a local package. First install wp-env locally as a dev dependency:

```
$ npm i @wordpress/env --save-dev
```

If you have also installed wp-env globally, running it will automatically execute the local, project-level package. Alternatively, you can execute wp-env via [npx](#), a utility automatically installed with npm. npx finds binaries like wp-env installed through node modules. As an example: `npx wp-env start --update`.

If you don't wish to use the global installation or npx, modify your package.json and add an extra command to npm scripts (<https://docs.npmjs.com/misc/scripts>):

```
"scripts": {
  "wp-env": "wp-env"
}
```

When installing wp-env in this way, all wp-env commands detailed in these docs must be prefixed with `npm run`, for example:

```
# You must add another double dash to pass flags to the script (wp-env) rather than npm
$ npm run wp-env start -- --update
```

instead of:

```
$ wp-env start --update
```

Usage

Starting the environment

First, ensure that Docker is running. You can do this by clicking on the Docker icon in the system tray or menu bar.

Then, change to a directory that contains a WordPress plugin or theme:

```
$ cd ~/gutenberg
```

Then, start the local environment:

```
$ wp-env start
```

Finally, navigate to `http://localhost:8888` in your web browser to see WordPress running with the local WordPress plugin or theme running and activated. Default login credentials are username: `admin` password: `password`.

Stopping the environment

To stop the local environment:

```
$ wp-env stop
```

Troubleshooting common problems

Many common problems can be fixed by running through the following troubleshooting steps in order:

1. Check that wp-env is running

First, check that `wp-env` is running. One way to do this is to have Docker print a table with the currently running containers:

```
$ docker ps
```

In this table, by default, you should see three entries: `wordpress` with port 8888, `tests-wordpress` with port 8889 and `mariadb` with port 3306.

2. Check the port number

By default `wp-env` uses port 8888, meaning that the local environment will be available at `http://localhost:8888`.

You can configure the port that `wp-env` uses so that it doesn't clash with another server by specifying the `WP_ENV_PORT` environment variable when starting `wp-env`:

```
$ WP_ENV_PORT=3333 wp-env start
```

Running `docker ps` and inspecting the `PORTS` column allows you to determine which port `wp-env` is currently using.

You may also specify the port numbers in your `.wp-env.json` file, but the environment variables will take precedence.

3. Restart wp-env with updates

Restarting `wp-env` will restart the underlying Docker containers which can fix many issues.

To restart `wp-env`, just run `wp-env start` again. It will automatically stop and start the container. If you also pass the `--update` argument, it will download updates and configure WordPress again.

```
$ wp-env start --update
```

4. Restart Docker

Restarting Docker will restart the underlying Docker containers and volumes which can fix many issues.

To restart Docker:

1. Click on the Docker icon in the system tray or menu bar.
2. Select *Restart*.

Once restarted, start `wp-env` again:

```
$ wp-env start
```

5. Reset the database

Resetting the database which the local environment uses can fix many issues, especially when they are related to the WordPress installation.

To reset the database:

⚠ WARNING: This will permanently delete any posts, pages, media, etc. in the local WordPress installation.

```
$ wp-env clean all  
$ wp-env start
```

6. Destroy everything and start again 🔥

When all else fails, you can use `wp-env destroy` to forcibly remove all of the underlying Docker containers, volumes, and files. This will allow you to start from scratch.

To do so:

⚠ WARNING: This will permanently delete any posts, pages, media, etc. in the local WordPress installation.

```
$ wp-env destroy  
# This new instance is a fresh start with no existing data:  
$ wp-env start
```

Using included WordPress PHPUnit test files

Out of the box `wp-env` includes the [WordPress' PHPUnit test files](#) corresponding to the version of WordPress installed. There is an environment variable, `WP_TESTS_DIR`, which points to the location of these files within each container. By including these files in the environment, we remove the need for you to use a package or install and mount them yourself. If you do not want

to use these files, you should ignore the `WP_TESTS_DIR` environment variable and load them from the location of your choosing.

Customizing the `wp-tests-config.php` file

While we do provide a default `wp-tests-config.php` file within the environment, there may be cases where you want to use your own. WordPress provides a `WP_TESTS_CONFIG_FILE_PATH` constant that you can use to change the `wp-config.php` file used for testing. Set this to a desired path in your `bootstrap.php` file and the file you've chosen will be used instead of the one included in the environment.

Using composer, phpunit, and wp-cli tools.

For ease of use, Composer, PHPUnit, and wp-cli are available for in the environment. To run these executables, use `wp-env run <env> <tool> <command>`. For example, `wp-env run cli composer install`, or `wp-env run tests-cli phpunit`. You can also access various shells like `wp-env run cli bash` or `wp-env run cli wp shell`.

For the `env` part, `cli` and `wordpress` share a database and mapped volumes, but more tools are available in the `cli` environment. You should use the `tests-cli`/`tests-wordpress` environments for a separate testing database.

By default, the cwd of the `run` command is the root of the WordPress install. If you're working on a plugin, you likely need to pass `--env-cwd` to make sure `composer/phpunit` commands are executed relative to the plugin you're working on. For example, `wp-env run cli --env-cwd=wp-content/plugins/gutenberg composer install`.

To make this easier, it's often helpful to add scripts in your `package.json` file:

```
{  
  "scripts": {  
    "composer": "wp-env run cli --env-cwd=wp-content/plugins/gutenberg  
  }  
}
```

Then, `npm run composer install` would run `composer install` in the environment. You could also do this for `phpunit`, `wp-cli`, etc.

Using Xdebug

Xdebug is installed in the `wp-env` environment, but it is turned off by default. To enable Xdebug, you can use the `--xdebug` flag with the `wp-env start` command. Here is a reference to how the flag works:

```
# Sets the Xdebug mode to "debug" (for step debugging):  
wp-env start --xdebug  
  
# Sets the Xdebug mode to "off":  
wp-env start  
  
# Enables each of the Xdebug modes listed:  
wp-env start --xdebug=profile,trace,debug
```

When you're running `wp-env` using `npm run`, like when working in the Gutenberg repo or when `wp-env` is a local project dependency, don't forget to add an extra double dash before the `--xdebug` command:

```
npm run wp-env start -- --xdebug
# Alternatively, use npx:
npx wp-env start --xdebug
```

If you forget about that, the `--xdebug` parameter will be passed to `npm` instead of the `wp-env start` command and it will be ignored.

You can see a reference on each of the Xdebug modes and what they do in the [Xdebug documentation](#).

Since we are only installing Xdebug 3, Xdebug is only supported for PHP versions greater than or equal to 7.2 (the default). Xdebug won't be installed if `phpVersion` is set to a legacy version.

[Xdebug IDE support](#)

To connect to Xdebug from your IDE, you can use these IDE settings. This bit of JSON was tested for VS Code's `launch.json` format (which you can [learn more about here](#)) along with [this PHP Debug extension](#). Its path mapping also points to a specific plugin — you should update this to point to the source you are working with inside of the `wp-env` instance.

You should only have to translate `port` and `pathMappings` to the format used by your own IDE.

```
{
  "name": "Listen for XDebug",
  "type": "php",
  "request": "launch",
  "port": 9003,
  "pathMappings": {
    "/var/www/html/wp-content/plugins/gutenberg": "${workspaceFolder}/"
  }
}
```

After you create a `.vscode/launch.json` file in your repository, you probably want to add it to your [global gitignore file](#) so that it stays private for you and is not committed to the repository.

Once your IDEs Xdebug settings have been enabled, you should just have to launch the debugger, put a breakpoint on any line of PHP code, and then refresh your browser!

Here is a summary:

1. Start `wp-env` with xdebug enabled: `wp-env start --xdebug`
2. Install a suitable Xdebug extension for your IDE if it does not include one already.
3. Configure the IDE debugger to use port `9003` and the correct source files in `wp-env`.
4. Launch the debugger and put a breakpoint on any line of PHP code.
5. Refresh the URL `wp-env` is running at and the breakpoint should trigger.

Command reference

`wp-env` creates generated files in the `wp-env` home directory. By default, this is `~/ .wp-env`. The exception is Linux, where files are placed at `~/wp-env` [for compatibility with Snap Packages](#). The `wp-env` home directory contains a subdirectory for each project named `/ $md5_of_project_path`. To change the `wp-env` home directory, set the `WP_ENV_HOME` environment variable. For example, running `WP_ENV_HOME="something" wp-env start` will download the project files to the directory `./something/ $md5_of_project_path` (relative to the current directory).

wp-env start

The `start` command installs and initializes the WordPress environment, which includes downloading any specified remote sources. By default, `wp-env` will not update or re-configure the environment except when the configuration file changes. Tell `wp-env` to update sources and apply the configuration options again with `wp-env start --update`. This will not overwrite any existing content.

`wp-env start`

Starts WordPress for development on port 8888 (`http://localhost:8888`) (override with `WP_ENV_PORT`) and tests on port 8889 (`http://localhost:8889`) (override with `WP_ENV_TESTS_PORT`). The current working directory must be a WordPress installation, a plugin, a theme, or contain a `.wp-env.json` file. first install, use the '`--update`' flag to download updates to mapped sourc to re-apply WordPress configuration options.

Options:

| | | |
|------------------------|---|----------------------------|
| <code>--debug</code> | Enable debug output. | [boolean] [default: false] |
| <code>--update</code> | Download source updates and apply WordPress configuration. | [boolean] [default: false] |
| <code>--xdebug</code> | Enables Xdebug. If not passed, Xdebug is turned off. If no modes are set, uses "debug". You may set multiple Xdebug modes by passing them in a comma-separated list: `--xdebug=develop,coverage`. https://xdebug.org/docs/all_settings#mode for information about Xdebug modes. | [string] |
| <code>--scripts</code> | Execute any configured lifecycle scripts. | [boolean] [default: true] |

wp-env stop

`wp-env stop`

Stops running WordPress for development and tests and frees the ports.

Options:

| | | |
|----------------------|----------------------|----------------------------|
| <code>--debug</code> | Enable debug output. | [boolean] [default: false] |
|----------------------|----------------------|----------------------------|

wp-env clean [environment]

`wp-env clean [environment]`

Cleans the WordPress databases.

Positionals:
environment Which environments' databases to clean.
[string] [choices: "all", "development", "tests"] [default: "t

Options:
--debug Enable debug output. [boolean] [default:
--scripts Execute any configured lifecycle scripts. [boolean] [default:

[wp-env run <container> \[command...\]](#)

The run command can be used to open shell sessions, invoke WP-CLI commands, or run any arbitrary commands inside of a container.

In some cases `wp-env run` may conflict with options that you are passing to the container. When this happens, `wp-env` will treat the option as its own and take action accordingly. For example, if you try `wp-env run cli php --help`, you will receive the `wp-env` help text.

You can get around this by passing any conflicting options after a double dash. `wp-env` will not process anything after the double dash and will simply pass it on to the container. To get the PHP help text you would use `wp-env run cli php -- --help`.

`wp-env run <container> [command...]`

Runs an arbitrary command in one of the underlying Docker containers. A double dash can be used to pass arguments to the container without parsing them. This is necessary if you are using an option that is defined below. You can use `bash` to open a shell session and both `composer` and `phpunit` are available in all WordPress and CLI containers. WP-CLI is also available in the CLI containers.

Positionals:
container The Docker service to run the command on.
[string] [required] [choices: "mysql", "tests-mysql", "wordpress", "tests-wordpress", "cli", "tests-cli", "composer", "phpunit"]
command The command to run. [req

Options:
--debug Enable debug output. [boolean] [default:
--env-cwd The command's working directory inside of the container. Paths without a leading slash are relative to the WordPress root.
[string] [default:

For example:

Displaying the users on the development instance:

```
wp-env run cli wp user list
: Running `wp user list` in 'cli'.
```

| ID | user_login | display_name | user_email | user_registered | ro |
|----|------------|--------------|------------|-----------------|----|
|----|------------|--------------|------------|-----------------|----|

```
1      admin    admin    wordpress@example.com  2020-03-05 10:45:14      ad
✓ Ran `wp user list` in 'cli'. (in 2s 374ms)
```

Creating a post on the tests instance:

```
wp-env run tests-cli "wp post create --post_type=page --post_title='Ready'
i Starting 'wp post create --post_type=page --post_title='Ready'' on the t
Success: Created post 5.
✓ Ran `wp post create --post_type=page --post_title='Ready'` in 'tests-cl
```

Opening the WordPress shell on the tests instance and running PHP commands:

```
wp-env run tests-cli wp shell
i Starting 'wp shell' on the tests-cli container. Exit the WordPress shell
Starting 31911d623e75f345e9ed328b9f48cff6_mysql_1 ... done
Starting 31911d623e75f345e9ed328b9f48cff6_tests-wordpress_1 ... done
wp> echo( 'hello world!' );
hello world!
wp> ^C
✓ Ran `wp shell` in 'tests-cli'. (in 16s 400ms)
```

Installing a plugin or theme on the development instance

```
wp-env run cli wp plugin install custom-post-type-ui

Creating 500cd328b649d63e882d5c4695871d04_cli_run ... done
Installing Custom Post Type UI (1.9.2)
Downloading installation package from https://downloads.wordpress.org/plug
The authenticity of custom-post-type-ui.zip could not be verified as no si
Unpacking the package...
Installing the plugin...
Plugin installed successfully.
Success: Installed 1 of 1 plugins.
✓ Ran `plugin install custom-post-type-ui` in 'cli'. (in 6s 483ms)
```

Changing the permalink structure

You might want to do this to enable access to the REST API (`wp-env/wp/v2/`) endpoint in your `wp-env` environment. The endpoint is not available with plain permalinks.

Examples

To set the permalink to just the post name:

```
wp-env run cli "wp rewrite structure '/%postname%/"
```

To set the permalink to the year, month, and post name:

```
wp-env run cli "wp rewrite structure '/%year%/%monthnum%/%postname%/"
```

[wp-env destroy](#)

```
wp-env destroy
```

Destroy the WordPress environment. Deletes docker containers, volumes, and networks associated with the WordPress environment and removes local files.

Options:

| | | |
|-----------|---|----------------------------|
| --debug | Enable debug output. | [boolean] [default: false] |
| --scripts | Execute any configured lifecycle scripts. | [boolean] [default: true] |

[wp-env logs \[environment\]](#)

```
wp-env logs
```

displays PHP and Docker logs for given WordPress environment.

Positionals:

| | |
|---|---|
| environment | Which environment to display the logs from. |
| [string] [choices: "development", "tests", "all"] | [default: "development"] |

Options:

| | | |
|---------|--------------------------------|----------------------------|
| --debug | Enable debug output. | [boolean] [default: false] |
| --watch | Watch for logs as they happen. | [boolean] [default: false] |

[wp-env install-path](#)

Get the path where all of the environment files are stored. This includes the Docker files, WordPress, PHPUnit files, and any sources that were downloaded.

Example:

```
$ wp-env install-path
```

```
/home/user/.wp-env/63263e6506becb7b8613b02d42280a49
```

[.wp-env.json](#)

You can customize the WordPress installation, plugins and themes that the development environment will use by specifying a `.wp-env.json` file in the directory that you run `wp-env` from.

`.wp-env.json` supports fields for options applicable to both the tests and development instances.

| Field | Type | Default | Description |
|--------------|----------------|---------|--|
| "core" | string \ null | | The WordPress installation to use. If <code>null</code> is specified, <code>wp-env</code> will use the latest production release of WordPress. |
| "phpVersion" | string \ null | | The PHP version to use. If <code>null</code> is specified, <code>wp-env</code> will use the |

| Field | Type | Default | Description |
|--------------|-------------|------------------------------------|--|
| "plugins" | string[] [] | | default version used with production release of WordPress. |
| "themes" | string[] [] | | A list of themes to install in the environment. |
| "port" | integer | 8888 (8889 for the tests instance) | The primary port number to use for the installation. You'll access the instance through the port: 'http://localhost:8888'. |
| "testsPort" | integer | 8889 | The port number for the test site. You'll access the instance through the port: 'http://localhost:8889'. |
| "config" | Object | See below. | Mapping of wp-config.php constants to their desired values. |
| "mappings" | Object | "{}" | Mapping of WordPress directories to local directories to be mounted in the WordPress instance. |

Note: the port number environment variables (WP_ENV_PORT and WP_ENV_TESTS_PORT) take precedent over the .wp-env.json values.

Several types of strings can be passed into the core, plugins, themes, and mappings fields.

| Type | Format | Example(s) |
|-------------------|--|--|
| Relative path | .<path>\ ~<path> | "../a/directory", "../a/directory", "~/a/directory" |
| Absolute path | /<path>\ <letter>:\<path> | "/a/directory", "C:\\a\\directory" |
| GitHub repository | <owner>/<repo>[#<ref>] | "WordPress/WordPress", "WordPress/gutenberg#trunk", if no branch is provided wp-env will fall back to the repos default branch |
| SSH repository | ssh://user@host/<owner>/<repo>.git[#<ref>] | "ssh://git@github.com/WordPress/WordPress.git" |
| ZIP File | http[s]://<host>/<path>.zip | "https://wordpress.org/wordpress-5.4-beta2.zip" |

Remote sources will be downloaded into a temporary directory located in ~/.wp-env.

Additionally, the key env is available to override any of the above options on an individual-environment basis. For example, take the following .wp-env.json file:

```
{
  "plugins": [ "." ],
  "config": {
    "KEY_1": true,
    "KEY_2": false
}
```

```

    },
    "env": {
        "development": {
            "themes": [ "./one-theme" ]
        },
        "tests": {
            "config": {
                "KEY_1": false
            },
            "port": 3000
        }
    }
}

```

On the development instance, `cwd` will be mapped as a plugin, `one-theme` will be mapped as a theme, `KEY_1` will be set to true, and `KEY_2` will be set to false. Also note that the default port, 8888, will be used as well.

On the tests instance, `cwd` is still mapped as a plugin, but no theme is mapped. Additionally, while `KEY_2` is still set to false, `KEY_1` is overridden and set to false. 3000 overrides the default port as well.

This gives you a lot of power to change the options applicable to each environment.

[.wp-env.override.json](#)

Any fields here will take precedence over `.wp-env.json`. This file is useful when ignored from version control, to persist local development overrides. Note that options like `plugins` and `themes` are not merged. As a result, if you set `plugins` in your override file, this will override all of the plugins listed in the base-level config. The only keys which are merged are `config` and `mappings`. This means that you can set your own `wp-config` values without losing any of the default values.

[Default wp-config values.](#)

On the development instance, these `wp-config` values are defined by default:

```

WP_DEBUG: true,
SCRIPT_DEBUG: true,
WP_PHP_BINARY: 'php',
WP_TESTS_EMAIL: 'admin@example.org',
WP_TESTS_TITLE: 'Test Blog',
WP_TESTS_DOMAIN: 'localhost',
WP_SITEURL: 'http://localhost',
WP_HOME: 'http://localhost',

```

On the test instance, all of the above are still defined, but `WP_DEBUG` and `SCRIPT_DEBUG` are set to false.

These can be overridden by setting a value within the `config` configuration. Setting it to `null` will prevent the constant being defined entirely.

Additionally, the values referencing a URL include the specified port for the given environment. So if you set `testsPort: 3000`, `port: 2000`, `WP_HOME` (for example) will be `http://localhost:3000`` on the `tests` instance and `http://localhost:2000`` on the development instance.

Lifecycle Scripts

Using the `lifecycleScripts` option in `.wp-env.json` will allow you to set arbitrary commands to be executed at certain points in the lifecycle. This configuration can also be overridden using `WP_ENV_LIFECYCLE_SCRIPT_{LIFECYCLE_EVENT}` environment variables, with the remainder being the all-caps snake_case name of the option, for example, `WP_ENV_LIFECYCLE_SCRIPT_AFTER_START`. Keep in mind that these will be executed on both fresh and existing environments, so, ensure any commands you build won't break on subsequent executions.

- `afterStart`: Runs after `wp-env start` has finished setting up the environment.
- `afterClean`: Runs after `wp-env clean` has finished cleaning the environment.
- `afterDestroy`: Runs after `wp-env destroy` has destroyed the environment.

Examples

Latest stable WordPress + current directory as a plugin

This is useful for plugin development.

```
{  
  "core": null,  
  "plugins": [ "." ]  
}
```

Latest development WordPress + current directory as a plugin

This is useful for plugin development when upstream Core changes need to be tested. This can also be set via the environment variable `WP_ENV_CORE`.

```
{  
  "core": "WordPress/WordPress#master",  
  "plugins": [ "." ]  
}
```

Local wordpress-develop + current directory as a plugin

This is useful for working on plugins and WordPress Core at the same time.

If you are running a *build* of `wordpress-develop`, point `core` to the `build` directory.

```
{  
  "core": ".../wordpress-develop/build",  
  "plugins": [ "." ]  
}
```

If you are running `wordpress-develop` in a dev mode (e.g. the watch command `dev` or the dev build `build:dev`), then point `core` to the `src` directory.

```
{  
  "core": "../wordpress-develop/src",  
  "plugins": [ "." ]  
}
```

A complete testing environment

This is useful for integration testing: that is, testing how old versions of WordPress and different combinations of plugins and themes impact each other.

```
{  
  "core": "WordPress/WordPress#5.2.0",  
  "plugins": [ "WordPress/wp-lazy-loading", "WordPress/classic-editor" ]  
  "themes": [ "WordPress/theme-experiments" ]  
}
```

Add mu-plugins and other mapped directories

You can add mu-plugins via the mapping config. The mapping config also allows you to mount a directory to any location in the wordpress install, so you could even mount a subdirectory. Note here that theme-1, will not be activated.

```
{  
  "plugins": [ "." ],  
  "mappings": {  
    "wp-content/mu-plugins": "./path/to/local/mu-plugins",  
    "wp-content/themes": "./path/to/local/themes",  
    "wp-content/themes/specific-theme": "./path/to/local/theme-1"  
  }  
}
```

Avoid activating plugins or themes on the instance

Since all plugins in the `plugins` key are activated by default, you should use the `mappings` key to avoid this behavior. This might be helpful if you have a test plugin that should not be activated all the time.

```
{  
  "plugins": [ "." ],  
  "mappings": {  
    "wp-content/plugins/my-test-plugin": "./path/to/test/plugin"  
  }  
}
```

Map a plugin only in the tests environment

If you need a plugin active in one environment but not the other, you can use `env.<envName>` to set options specific to one environment. Here, we activate cwd and a test plugin on the tests instance. This plugin is not activated on any other instances.

```
{
  "plugins": [ "." ],
  "env": {
    "tests": {
      "plugins": [ ".", "path/to/test/plugin" ]
    }
  }
}
```

Custom Port Numbers

You can tell `wp-env` to use a custom port number so that your instance does not conflict with other `wp-env` instances.

```
{
  "plugins": [ "." ],
  "port": 4013,
  "env": {
    "tests": {
      "port": 4012
    }
  }
}
```

Specific PHP Version

You can tell `wp-env` to use a specific PHP version for compatibility and testing. This can also be set via the environment variable `WP_ENV_PHP_VERSION`.

```
{
  "phpVersion": "7.2",
  "plugins": [ "." ]
}
```

Node Lifecycle Script

This is useful for performing some actions after setting up the environment, such as bootstrapping an E2E test environment.

```
{
  "lifecycleScripts": {
    "afterStart": "node tests/e2e/bin/setup-env.js"
  }
}
```

Advanced PHP settings

You can set PHP settings by mapping an `.htaccess` file. This maps an `.htaccess` file to the WordPress root (`/var/www/html`) from the directory in which you run `wp-env`.

```
{
  "mappings": {
    ".htaccess": ".htaccess"
```

```
}
```

Then, your .htaccess file can contain various settings like this:

```
# Note: the default upload value is 1G.  
php_value post_max_size 2G  
php_value upload_max_filesize 2G  
php_value memory_limit 2G
```

This is useful if there are options you'd like to add to `php.ini`, which is difficult to access in this environment.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/env](#)

[Previous @wordpress/element](#) [Previous: @wordpress/element](#)
[Next @wordpress/escape-html](#) [Next: @wordpress/escape-html](#)

@wordpress/escape-html

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [escapeAmpersand](#)
 - [escapeAttribute](#)
 - [escapeEditableHTML](#)
 - [escapeHTML](#)
 - [escapeLessThan](#)
 - [escapeQuotationMark](#)

- [isValidAttributeName](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Escape HTML utils.

[Installation](#)

Install the module

```
npm install @wordpress/escape-html
```

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](#)

[escapeAmpersand](#)

Returns a string with ampersands escaped. Note that this is an imperfect implementation, where only ampersands which do not appear as a pattern of named, decimal, or hexadecimal character references are escaped. Invalid named references (i.e. ambiguous ampersand) are still permitted.

Related

- <https://w3c.github.io/html/syntax.html#character-references>
- <https://w3c.github.io/html/syntax.html#ambiguous-ampersand>
- <https://w3c.github.io/html/syntax.html#named-character-references>

Parameters

- `value string`: Original string.

Returns

- `string`: Escaped string.

[escapeAttribute](#)

Returns an escaped attribute value.

Related

- <https://w3c.github.io/html/syntax.html#elements-attributes> “[...] the text cannot contain an ambiguous ampersand [...] must not contain any literal U+0022 QUOTATION MARK characters (“”)

Note we also escape the greater than symbol, as this is used by wptexturize to split HTML strings. This is a WordPress specific fix

Note that if a resolution for Trac#45387 comes to fruition, it is no longer necessary for `__unstableEscapeGreaterThan` to be used.

See: <https://core.trac.wordpress.org/ticket/45387>

Parameters

- `value string`: Attribute value.

Returns

- `string`: Escaped attribute value.

[escapeEditableHTML](#)

Returns an escaped Editable HTML element value. This is different from `escapeHTML`, because for editable HTML, ALL ampersands must be escaped in order to render the content correctly on the page.

Parameters

- `value string`: Element value.

Returns

- `string`: Escaped HTML element value.

[escapeHTML](#)

Returns an escaped HTML element value.

Related

- <https://w3c.github.io/html/syntax.html#writing-html-documents-elements> “the text must not contain the character U+003C LESS-THAN SIGN (<) or an ambiguous ampersand.”

Parameters

- `value string`: Element value.

Returns

- `string`: Escaped HTML element value.

[escapeLessThan](#)

Returns a string with less-than sign replaced.

Parameters

- `value string`: Original string.

Returns

- `string`: Escaped string.

[escapeQuotationMark](#)

Returns a string with quotation marks replaced.

Parameters

- `value string`: Original string.

Returns

- `string`: Escaped string.

[isValidAttributeName](#)

Returns true if the given attribute name is valid, or false otherwise.

Parameters

- `name string`: Attribute name to test.

Returns

- `boolean`: Whether attribute is valid.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/escape-html”](#)

[Previous @wordpress/env](#) Previous: [@wordpress/env](#)

Next [@wordpress/eslint-plugin](#) Next: [@wordpress/eslint-plugin](#)

@wordpress/eslint-plugin

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
 - [Rulesets](#)
 - [Rules](#)
 - [Legacy](#)
- [Contributing to this package](#)

[↑ Back to top](#)

[ESLint](#) plugin including configurations and custom rules for WordPress development.

[Installation](#)

Install the module

```
npm install @wordpress/eslint-plugin --save-dev
```

Note: This package requires node 14.0.0 or later, and npm 6.14.4 or later. It is not compatible with older versions.

[Usage](#)

To opt-in to the default configuration, extend your own project's `.eslintrc` file:

```
{  
  "extends": [ "plugin:@wordpress/eslint-plugin/recommended" ]  
}
```

Refer to the [ESLint documentation on Shareable Configs](#) for more information.

The recommended preset will include rules governing an ES2015+ environment, and includes rules from the [eslint-plugin-jsdoc](#), [eslint-plugin-jsx-a11y](#), [eslint-plugin-react](#), and other similar plugins.

This preset offers an optional integration with the [eslint-plugin-prettier](#) package that runs [Prettier](#) code formatter and reports differences as individual ESLint issues. You can activate it by installing the [prettier](#) package separately with:

```
npm install prettier --save-dev
```

Finally, this ruleset also includes an optional integration with the [@typescript-eslint/eslint-plugin](#) package that enables ESLint to support [TypeScript](#) language. You can activate it by installing the [typescript](#) package separately with:

```
npm install typescript --save-dev
```

There is also `recommended-with-formatting` ruleset for projects that want to ensure that [Prettier](#) and [TypeScript](#) integration is never activated. This preset has the native ESLint code formatting rules enabled instead.

[Rulesets](#)

Alternatively, you can opt-in to only the more granular rulesets offered by the plugin. These include:

- `custom` – custom rules for WordPress development.
- `es5` – rules for legacy ES5 environments.
- `esnext` – rules for ES2015+ environments.
- `i18n` – rules for internationalization.
- `jsdoc` – rules for JSDoc comments.
- `jsx-a11y` – rules for accessibility in JSX.
- `react` – rules for React components.
- `test-e2e` – rules for end-to-end tests written in Puppeteer.
- `test-unit` – rules for unit tests written in Jest.
- `test-playwright` – rules for end-to-end tests written in Playwright.

For example, if your project does not use React, you could consider extending including only the `ESNext` rules in your project using the following `extends` definition:

```
{  
  "extends": [ "plugin:@wordpress/eslint-plugin/esnext" ]  
}
```

These rules can be used additively, so you could extend both `esnext` and `custom` rulesets, but omit the `react` and `jsx-a11y` configurations.

The granular rulesets will not define any environment globals. As such, if they are required for your project, you will need to define them yourself.

[Rules](#)

| Rule | Description | Recommended |
|---|---|-------------|
| data-no-store-string-literals | Discourage passing string literals to reference data stores | |
| dependency-group | Enforce dependencies docblocks formatting | ✓ |
| is-gutenberg-plugin | Governs the use of the <code>process.env.IS_GUTENBERG_PLUGIN</code> constant | ✓ |
| no-base-control-with-label-without-id | Disallow the usage of <code>BaseControl</code> component with a <code>label</code> prop set but omitting the <code>id</code> property | ✓ |
| no-unguarded-get-range-at-calls | Disallow the usage of unguarded <code>getRangeAt</code> calls | ✓ |
| no-unsafe-wp-apis | | ✓ |

| Rule | Description | Recommended |
|--|---|-------------|
| no-unused-vars-before-return | Disallow the usage of unsafe APIs from @wordpress/* packages | |
| react-no-unsafe-timeout | Disallow assigning variable values if unused before a return | ✓ |
| valid-sprintf | Disallow unsafe setTimeout in component | |
| i18n-ellipsis | Enforce valid sprintf usage | ✓ |
| i18n-no-collapsible-whitespace | Disallow using three dots in translatable strings | ✓ |
| i18n-no-placeholders-only | Disallow collapsible whitespace in translatable strings | ✓ |
| i18n-no-variables | Prevent using only placeholders in translatable strings | ✓ |
| i18n-text-domain | Enforce string literals as translation function arguments | ✓ |
| i18n-translator-comments | Enforce passing valid text domains | ✓ |
| i18n-no-flanking-whitespace | Enforce adding translator comments | ✓ |
| i18n-no-flanking-whitespace | Disallow leading or trailing whitespace in translatable strings | |
| i18n-hyphenated-range | Disallow hyphenated numerical ranges in translatable strings | |

Legacy

If you are using WordPress' `.jshintrc` JSHint configuration and you would like to take the first step to migrate to an ESLint equivalent it is also possible to define your own project's `.eslintrc` file as:

```
{
  "extends": [ "plugin:@wordpress/eslint-plugin/jshint" ]
}
```

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/eslint-plugin”](#)

[Previous @wordpress/escape-html](#) [Previous: @wordpress/escape-html](#)

[Next @wordpress/format-library](#) [Next: @wordpress/format-library](#)

@wordpress/format-library

In this article

[Table of Contents](#)

- [Installation](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Format library for the WordPress editor.

[Installation](#)

Install the module

```
npm install @wordpress/format-library --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.*

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/format-library”](#)

[Previous @wordpress/eslint-plugin](#) Previous: [@wordpress/eslint-plugin](#)

Next [@wordpress/hooks](#) Next: [@wordpress/hooks](#)

@wordpress/hooks

In this article

Table of Contents

- [Installation](#)
 - [Usage](#)
 - [API Usage](#)
 - [Events on action/filter add or remove](#)
 - [The all hook](#)
- [Contributing to this package](#)

[↑ Back to top](#)

A lightweight & efficient EventManager for JavaScript.

[Installation](#)

Install the module

```
npm install @wordpress/hooks --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.*

[Usage](#)

In your JavaScript project, use hooks as follows:

```
import { createHooks } from '@wordpress/hooks';

myObject.hooks = createHooks();
myObject.hooks.addAction(); //etc...
```

The global instance

In the above example, we are creating a custom instance of the `Hooks` object and registering hooks there. The package also creates a default global instance that's accessible through the `defaultHooks` named exports, and its methods are also separately exported one-by-one.

In the WordPress context, that enables API functions to be called via the global `wp.hooks` object, like `wp.hooks.addAction()`, etc.

One notable difference between the JS and PHP hooks API is that in the JS version, `addAction()` and `addFilter()` also need to include a namespace as the second argument. Namespace uniquely identifies a callback in the form `vendor/plugin/function`.

API Usage

- `createHooks()`
- `addAction('hookName', 'namespace', callback, priority)`
- `addFilter('hookName', 'namespace', callback, priority)`
- `removeAction('hookName', 'namespace')`
- `removeFilter('hookName', 'namespace')`
- `removeAllActions('hookName')`
- `removeAllFilters('hookName')`
- `doAction('hookName', arg1, arg2, moreArgs, finalArg)`
- `applyFilters('hookName', content, arg1, arg2, moreArgs, finalArg)`
- `doingAction('hookName')`
- `doingFilter('hookName')`
- `didAction('hookName')`
- `didFilter('hookName')`
- `hasAction('hookName', 'namespace')`
- `hasFilter('hookName', 'namespace')`
- `actions`
- `filters`
- `defaultHooks`

Events on action/filter add or remove

Whenever an action or filter is added or removed, a matching `hookAdded` or `hookRemoved` action is triggered.

- `hookAdded` action is triggered when `addFilter()` or `addAction()` method is called, passing values for `hookName`, `functionName`, `callback` and `priority`.
- `hookRemoved` action is triggered when `removeFilter()` or `removeAction()` method is called, passing values for `hookName` and `functionName`.

The all hook

In non-minified builds developers can register a filter or action that will be called on *all* hooks, for example: `addAction('all', 'namespace', callbackFunction);`. Useful for debugging, the code supporting the `all` hook is stripped from the production code for performance reasons.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/hooks](#)

[Previous @wordpress/format-library](#) [Previous: @wordpress/format-library](#)
[Next @wordpress/html-entities](#) [Next: @wordpress/html-entities](#)

@wordpress/html-entities

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [decodeEntities](#)
- [Contributing to this package](#)

[↑ Back to top](#)

HTML entity utilities for WordPress.

Installation

Install the module

```
npm install @wordpress/html-entities --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

decodeEntities

Decodes the HTML entities from a given string.

Usage

```
const result = decodeEntities( '&aacute;' );
console.log( result ); // result will be "á"
```

Parameters

- *html string*: String that contain HTML entities.

Returns

- *string*: The decoded string.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/html-entities”](#)

[Previous](#) [@wordpress/hooks](#) [Previous: @wordpress/hooks](#)
[Next](#) [@wordpress/i18n](#) [Next: @wordpress/i18n](#)

@wordpress/i18n

In this article

Table of Contents

- [Installation](#)
- [Usage](#)

- [API](#)
 - [createI18n](#)
 - [defaultI18n](#)
 - [getLocaleData](#)
 - [hasTranslation](#)
 - [isRTL](#)
 - [resetLocaleData](#)
 - [setLocaleData](#)
 - [sprintf](#)
 - [subscribe](#)
 - [_n](#)
 - [_nx](#)
 - [_x](#)
 - [__](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Internationalization utilities for client-side localization. See [How to Internationalize Your Plugin](#) for server-side documentation.

[Installation](#)

Install the module:

```
npm install @wordpress/i18n --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.*

[Usage](#)

```
import { sprintf, _n } from '@wordpress/i18n';

sprintf( _n( '%d hat', '%d hats', 4, 'text-domain' ), 4 );
// 4 hats
```

For a complete example, see the [Internationalization section of the Block Editor Handbook](#).

[API](#)

[createI18n](#)

Create an i18n instance

Parameters

- *initialData* [`LocaleData`]: Locale data configuration.
- *initialDomain* [`string`]: Domain for which configuration applies.
- *hooks* [`Hooks`]: Hooks implementation.

Returns

- `I18n`: I18n instance.

[defaultI18n](#)

Default, singleton instance of `I18n`.

[getLocaleData](#)

Returns locale data by domain in a Jed-formatted JSON object shape.

Related

- <http://messageformat.github.io/Jed/>

Parameters

- `domain [string]`: Domain for which to get the data.

Returns

- `LocaleData`: Locale data.

[hasTranslation](#)

Check if there is a translation for a given string (in singular form).

Parameters

- `single string`: Singular form of the string to look up.
- `context [string]`: Context information for the translators.
- `domain [string]`: Domain to retrieve the translated text.

Returns

- `boolean`: Whether the translation exists or not.

[isRTL](#)

Check if current locale is RTL.

RTL (Right To Left) is a locale property indicating that text is written from right to left. For example, the `he` locale (for Hebrew) specifies right-to-left. Arabic (`ar`) is another common language written RTL. The opposite of RTL, LTR (Left To Right) is used in other languages, including English (`en`, `en-US`, `en-GB`, etc.), Spanish (`es`), and French (`fr`).

Returns

- `boolean`: Whether locale is RTL.

[resetLocaleData](#)

Resets all current Tannin instance locale data and sets the specified locale data for the domain.
Accepts data in a Jed-formatted JSON object shape.

Related

- <http://messageformat.github.io/Jed/>

Parameters

- *data* [`LocaleData`]: Locale data configuration.
- *domain* [`string`]: Domain for which configuration applies.

[setLocaleData](#)

Merges locale data into the Tannin instance by domain. Accepts data in a Jed-formatted JSON object shape.

Related

- <http://messageformat.github.io/Jed/>

Parameters

- *data* [`LocaleData`]: Locale data configuration.
- *domain* [`string`]: Domain for which configuration applies.

[sprintf](#)

Returns a formatted string. If an error occurs in applying the format, the original format string is returned.

Related

- <https://www.npmjs.com/package/sprintf-js>

Parameters

- *format string*: The format of the string to generate.
- *args . . . **: Arguments to apply to the format.

Returns

- `string`: The formatted string.

[subscribe](#)

Subscribes to changes of locale data

Parameters

- *callback* `SubscribeCallback`: Subscription callback

Returns

- `UnsubscribeCallback`: Unsubscribe callback

n

Translates and retrieves the singular or plural form based on the supplied number.

Related

- https://developer.wordpress.org/reference/functions/_n/

Parameters

- `single string`: The text to be used if the number is singular.
- `plural string`: The text to be used if the number is plural.
- `number number`: The number to compare against to use either the singular or plural form.
- `domain [string]`: Domain to retrieve the translated text.

Returns

- `string`: The translated singular or plural form.

nx

Translates and retrieves the singular or plural form based on the supplied number, with gettext context.

Related

- https://developer.wordpress.org/reference/functions/_nx/

Parameters

- `single string`: The text to be used if the number is singular.
- `plural string`: The text to be used if the number is plural.
- `number number`: The number to compare against to use either the singular or plural form.
- `context string`: Context information for the translators.
- `domain [string]`: Domain to retrieve the translated text.

Returns

- `string`: The translated singular or plural form.

x

Retrieve translated string with gettext context.

Related

- https://developer.wordpress.org/reference/functions/_x/

Parameters

- `text string`: Text to translate.

- *context string*: Context information for the translators.
- *domain [string]*: Domain to retrieve the translated text.

Returns

- *string*: Translated context string without pipe.

—
Retrieve the translation of text.

Related

- https://developer.wordpress.org/reference/functions/_/

Parameters

- *text string*: Text to translate.
- *domain [string]*: Domain to retrieve the translated text.

Returns

- *string*: Translated text.

[**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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/i18n”](#)

[Previous](#) [@wordpress/html-entities](#) [Previous: @wordpress/html-entities](#)
[Next](#) [@wordpress/icons](#) [Next: @wordpress/icons](#)

@wordpress/icons

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Props](#)
- [Docs & Examples](#)
- [Contributing to this package](#)

[↑ Back to top](#)

WordPress Icons Library.

Installation

Install the module:

```
npm install @wordpress/icons --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.*

Usage

```
import { Icon, check } from '@wordpress/icons';

<Icon icon={ check } />;
```

Props

| Name | Type | Default | Description |
|------|---------|---------|-------------------------|
| size | integer | 24 | Size of icon in pixels. |

Docs & Examples

You can browse the icons docs and examples at <https://wordpress.github.io/gutenberg/?path=/docs/icons-icon-default>

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/icons”](#)

[Previous @wordpress/i18n](#) [Previous: @wordpress/i18n](#)

[Next @wordpress/interactivity-router](#) [Next: @wordpress/interactivity-router](#)

@wordpress/interactivity-router

In this article

[Table of Contents](#)

- [Usage](#)
- [Frequently Asked Questions](#)
 - [What is this?](#)
 - [Can I use it?](#)
 - [How do I get started?](#)
 - [Where can I ask questions?](#)
 - [Where can I share my feedback about the API?](#)
- [Installation](#)
- [Docs & Examples](#)

[↑ Back to top](#)

Note

This package is a extension of the API shared at [Proposal: The Interactivity API – A better developer experience in building interactive blocks](#). As part of an [Open Source project](#) we encourage participation in helping shape this API and the [discussions in GitHub](#) is the best place to engage.

This package defines an Interactivity API store with the `core/router` namespace, exposing state and actions like `navigate` and `prefetch` to handle client-side navigations.

[Usage](#)

The package is intended to be imported dynamically in the `view.js` files of interactive blocks.

```
import { store } from '@wordpress/interactivity';

store( 'myblock', {
    actions: {
        *navigate( e ) {
            e.preventDefault();
            const { actions } = yield import(
                '@wordpress/interactivity-router'
            );
            yield actions.navigate( e.target.href );
        },
    },
} );
```

Frequently Asked Questions

At this point, some of the questions you have about the Interactivity API may be:

What is this?

This is the base of a new standard to create interactive blocks. Read [the proposal](#) to learn more about this.

Can I use it?

You can test it, but it's still very experimental.

How do I get started?

The best place to start with the Interactivity API is this [Getting started guide](#). There you'll will find a very quick start guide and the current requirements of the Interactivity API.

Where can I ask questions?

The “[Interactivity API](#)” category in Gutenberg repo discussions is the best place to ask questions about the Interactivity API.

Where can I share my feedback about the API?

The “[Interactivity API](#)” category in Gutenberg repo discussions is also the best place to share your feedback about the Interactivity API.

Installation

Install the module:

```
npm install @wordpress/interactivity --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.*

Docs & Examples

[Interactivity API Documentation](#) is the best place to learn about this proposal. Although it's still in progress, some key pages are already available:

- [Getting Started Guide](#): Follow this Getting Started guide to learn how to scaffold a new project and create your first interactive blocks.
- [API Reference](#): Check this page for technical detailed explanations and examples of the directives and the store.

Here you have some more resources to learn/read more about the Interactivity API:

- [Interactivity API Discussions](#)
- [Proposal: The Interactivity API – A better developer experience in building interactive blocks](#)
- Developer Hours sessions ([Americas](#) & [APAC/EMEA](#))
- [wpmovies.dev](#) demo and its [wp-movies-demo](#) repo

First published

January 24, 2024

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/interactivity-router”](#)

[Previous @wordpress/icons](#) Previous: [@wordpress/icons](#)
[Next @wordpress/interactivity](#) Next: [@wordpress/interactivity](#)

@wordpress/interactivity

In this article

Table of Contents

- [Frequently Asked Questions](#)
 - [What is this?](#)
 - [Can I use it?](#)
 - [How do I get started?](#)
 - [Where can I ask questions?](#)
 - [Where can I share my feedback about the API?](#)
- [Installation](#)
- [Docs & Examples](#)

[↑ Back to top](#)

Warning

This package is only available in Gutenberg at the moment and not in WordPress Core as it is still very experimental, and very likely to change.

Note

This package enables the API shared at [Proposal: The Interactivity API – A better developer experience in building interactive blocks](#). As part of an [Open Source project](#) we encourage participation in helping shape this API and the [discussions in GitHub](#) is the best place to engage.

This package can be tested, but it's still very experimental.

The Interactivity API is [being used in some core blocks](#) but its use is still very limited.

Frequently Asked Questions

At this point, some of the questions you have about the Interactivity API may be:

What is this?

This is the base of a new standard to create interactive blocks. Read [the proposal](#) to learn more about this.

Can I use it?

You can test it, but it's still very experimental.

How do I get started?

The best place to start with the Interactivity API is this [Getting started guide](#). There you'll will find a very quick start guide and the current requirements of the Interactivity API.

Where can I ask questions?

The “[Interactivity API” category](#) in Gutenberg repo discussions is the best place to ask questions about the Interactivity API.

Where can I share my feedback about the API?

The “[Interactivity API” category](#) in Gutenberg repo discussions is also the best place to share your feedback about the Interactivity API.

Installation

Install the module:

```
npm install @wordpress/interactivity --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.*

Docs & Examples

[Interactivity API Documentation](#) is the best place to learn about this proposal. Although it's still in progress, some key pages are already available:

- [Getting Started Guide](#): Follow this Getting Started guide to learn how to scaffold a new project and create your first interactive blocks.
- [API Reference](#): Check this page for technical detailed explanations and examples of the directives and the store.

Here you have some more resources to learn/read more about the Interactivity API:

- [Interactivity API Discussions](#)
- [Proposal: The Interactivity API – A better developer experience in building interactive blocks](#)
- Developer Hours sessions ([Americas](#) & [APAC/EMEA](#))
- [wpmovies.dev](#) demo and its [wp-movies-demo](#) repo

First published

June 28, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/interactivity”](#)

Previous [@wordpress/interactivity-router](#) Previous: [@wordpress/interactivity-router](#)
Next [@wordpress/interface](#) Next: [@wordpress/interface](#)

@wordpress/interface

In this article

Table of Contents

- [Installation](#)
- [API Usage](#)
 - [Complementary Areas](#)
 - [Pinned Items](#)
 - [Preferences](#)
- [Contributing to this package](#)

[↑ Back to top](#)

The Interface Package contains the basis to start a new WordPress screen as Edit Post or Edit Site. The package offers a data store and a set of components. The store is useful to contain common data required by a screen (e.g., active areas). The information is persisted across screen reloads.

The components allow one to implement functionality like a sidebar or menu items. Third-party plugins can extend them by default.

Installation

Install the module

```
npm install @wordpress/interface --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 Usage

Complementary Areas

This component was named after a [complementary landmark](#) – a supporting section of the document, designed to be complementary to the main content at a similar level in the DOM hierarchy, but remains meaningful when separated from the main content.

ComplementaryArea and ComplementaryArea.Slot form a slot fill pair to render complementary areas. Multiple ComplementaryArea components representing different complementary areas may be rendered at the same time, but only one appears on the slot depending on which complementary area is enabled.

It is possible to control which complementary is enabled by using the store:

Below are some examples of how to control the active complementary area using the store:

```
wp.data
  .select( 'core/interface' )
  .getActiveComplementaryArea( 'core/edit-post' );
// -> "edit-post/document"

wp.data
  .dispatch( 'core/interface' )
  .enableComplementaryArea( 'core/edit-post', 'edit-post/block' );

wp.data
  .select( 'core/interface' )
  .getActiveComplementaryArea( 'core/edit-post' );
// -> "edit-post/block"

wp.data
  .dispatch( 'core/interface' )
  .disableComplementaryArea( 'core/edit-post' );

wp.data
  .select( 'core/interface' )
  .getActiveComplementaryArea( 'core/edit-post' );
// -> null
```

Pinned Items

PinnedItems and PinnedItems.Slot form a slot fill pair to render pinned items (or areas) that act as a list of favorites similar to browser extensions listed in the Chrome Menu.

Example usage: ComplementaryArea component makes use of PinnedItems and automatically adds a pinned item for the complementary areas marked as a favorite.

```
wp.data.select( 'core/interface' ).isItemPinned( 'core/edit-post', 'edit-post' )
// -> false

wp.data.dispatch( 'core/interface' ).pinItem( 'core/edit-post', 'edit-post' )

wp.data.select( 'core/interface' ).isItemPinned( 'core/edit-post', 'edit-post' )
// -> true

wp.data.dispatch( 'core/interface' ).unpinItem( 'core/edit-post', 'edit-post' )

wp.data.select( 'core/interface' ).isItemPinned( 'core/edit-post', 'edit-post' )
```

Preferences

The interface package provides some helpers for implementing editor preferences.

Features

Features are boolean values used for toggling specific editor features on or off.

Set the default values for any features on editor initialization:

```
import { dispatch } from '@wordpress/data';
import { store as interfaceStore } from '@wordpress/interface';

function initialize() {
    // ...

    dispatch( interfaceStore ).setFeatureDefaults(
        'namespace/editor-or-plugin-name',
        {
            myFeatureName: true,
        }
    );

    // ...
}
```

Use the `toggleFeature` action and the `isFeatureActive` selector to toggle features within your app:

```
wp.data
    .select( 'core/interface' )
    .isFeatureActive( 'namespace/editor-or-plugin-name', 'myFeatureName' )
wp.data
    .dispatch( 'core/interface' )
```

```
.toggleFeature( 'namespace/editor-or-plugin-name' , 'myFeatureName' );
wp.data
  .select( 'core/interface' )
  .isFeatureActive( 'namespace/editor-or-plugin-name' , 'myFeatureName' )
```

The `MoreMenuDropdown` and `MoreMenuFeatureToggle` components help to implement an editor menu for changing preferences and feature values.

```
function MyEditorMenu() {
  return (
    <MoreMenuDropdown>
      { () => (
        <MenuGroup label={ __( 'Features' ) }>
          <MoreMenuFeatureToggle
            scope="namespace/editor-or-plugin-name"
            feature="myFeatureName"
            label={ __( 'My feature' ) }
            info={ __( 'A really awesome feature' ) }
            messageActivated={ __( 'My feature activated' ) }
            messageDeactivated={ __( 'My feature deactivated' ) }
          />
        </MenuGroup>
      ) }
    </MoreMenuDropdown>
  );
}
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/interface](#)

[Previous: @wordpress/interactivity](#) [Previous: @wordpress/interactivity](#)
[Next: @wordpress/is-shallow-equal](#) [Next: @wordpress/is-shallow-equal](#)

@wordpress/is-shallow-equal

In this article

Table of Contents

- [Usage](#)
- [Rationale](#)
- [Benchmarks](#)
- [Contributing to this package](#)

[↑ Back to top](#)

A function for performing a shallow comparison between two objects or arrays. Two values have shallow equality when all of their members are strictly equal to the corresponding member of the other.

Usage

The default export of @wordpress/is-shallow-equal is a function which accepts two objects or arrays:

```
import isShallowEqual from '@wordpress/is-shallow-equal';

isShallowEqual( { a: 1 }, { a: 1, b: 2 } );
// => false

isShallowEqual( { a: 1 }, { a: 1 } );
// => true

isShallowEqual( [ 1 ], [ 1, 2 ] );
// => false

isShallowEqual( [ 1 ], [ 1 ] );
// => true
```

You can import a specific implementation if you already know the types of values you are working with:

```
import { isShallowEqualArrays } from '@wordpress/is-shallow-equal';
import { isShallowEqualObjects } from '@wordpress/is-shallow-equal';
```

Shallow comparison differs from deep comparison by the fact that it compares members from each as being strictly equal to the other, meaning that arrays and objects will be compared by their *references*, not by their values (see also [Object Equality in JavaScript](#).) In situations where nested objects must be compared by value, consider using [fast-deep-equal](#) instead.

```
import isShallowEqual from '@wordpress/is-shallow-equal';
import fastDeepEqual from 'fast-deep-equal/es6'; // deep comparison

let object = { a: 1 };
```

```

isShallowEqual( [ { a: 1 } ], [ { a: 1 } ] );
// => false

fastDeepEqual( [ { a: 1 } ], [ { a: 1 } ] );
// => true

isShallowEqual( [ object ], [ object ] );
// => true

```

Rationale

Shallow equality utilities are already a dime a dozen. Since these operations are often at the core of critical hot code paths, the WordPress contributors had specific requirements that were found to only be partially satisfied by existing solutions.

In particular, it should...

1. ...consider non-primitive yet referentially-equal members values as equal.
 - Eliminates [is-equal-shallow](#) as an option.
2. ...offer a single function through which to interface, regardless of value type.
 - Eliminates [shallow-equal](#) as an option.
3. ...be barebones; only providing the basic functionality of shallow equality.
 - Eliminates [shallow-equals](#) as an option.
4. ...anticipate and optimize for referential sameness as equal.
 - Eliminates [is-equal-shallow](#) and [shallow-equals](#) as options.
5. ...be intended for use in non-Facebook projects.
 - Eliminates [fbjs/lib/shallowEqual](#) as an option.
6. ...be the most performant implementation.
 - See [Benchmarks](#).

Benchmarks

The following results were produced under Node v10.15.3 (LTS) on a MacBook Pro (Late 2016) 2.9 GHz Intel Core i7.

```

@wordpress/is-shallow-equal (type specific) (object,
equal) x 4,519,009 ops/sec ±1.09% (90 runs sampled)
>@wordpress/is-shallow-equal (type specific) (object,
same) x 795,527,700 ops/sec ±0.24% (93 runs sampled)
>@wordpress/is-shallow-equal (type specific) (object,
unequal) x 4,841,640 ops/sec ±0.94% (93 runs sampled)
>@wordpress/is-shallow-equal (type specific) (array,
equal) x 106,393,795 ops/sec ±0.16% (94 runs sampled)
>@wordpress/is-shallow-equal (type specific) (array,
same) x 800,741,511 ops/sec ±0.22% (95 runs sampled)
>@wordpress/is-shallow-equal (type specific) (array,
unequal) x 49,178,977 ops/sec ±1.99% (82 runs sampled)

@wordpress/is-shallow-equal (object, equal) x 4,449,367
ops/sec ±0.31% (91 runs sampled) >@wordpress/is-shallow-
equal (object, same) x 796,677,179 ops/sec ±0.23% (94
runs sampled) >@wordpress/is-shallow-equal (object,

```

unequal) x 4,989,529 ops/sec $\pm 0.30\%$ (91 runs sampled)
>@wordpress/is-shallow-equal (array, equal) x 44,840,546
ops/sec $\pm 1.18\%$ (89 runs sampled) @wordpress/is-shallow-
equal (array, same) x 794,344,723 ops/sec $\pm 0.24\%$ (91 runs
sampled) @wordpress/is-shallow-equal (array, unequal) x
49,860,115 ops/sec $\pm 1.73\%$ (85 runs sampled)

shallowequal (object, equal) x 3,702,126 ops/sec $\pm 0.87\%$
(92 runs sampled) >shallowequal (object, same) x
796,649,597 ops/sec $\pm 0.21\%$ (92 runs sampled)
>shallowequal (object, unequal) x 4,027,885 ops/sec
 $\pm 0.31\%$ (96 runs sampled) >shallowequal (array, equal) x
1,684,977 ops/sec $\pm 0.37\%$ (94 runs sampled) >shallowequal
(array, same) x 794,287,091 ops/sec $\pm 0.26\%$ (91 runs
sampled) >shallowequal (array, unequal) x 1,738,554 ops/
sec $\pm 0.29\%$ (91 runs sampled)

shallow-equal (type specific) (object, equal) x 4,669,656
ops/sec $\pm 0.34\%$ (92 runs sampled) >shallow-equal (type
specific) (object, same) x 799,610,214 ops/sec $\pm 0.20\%$ (95
runs sampled) >shallow-equal (type specific) (object,
unequal) x 4,908,591 ops/sec $\pm 0.49\%$ (93 runs sampled)
>shallow-equal (type specific) (array, equal) x
104,711,254 ops/sec $\pm 0.65\%$ (91 runs sampled) >shallow-
equal (type specific) (array, same) x 798,454,281 ops/sec
 $\pm 0.29\%$ (94 runs sampled) >shallow-equal (type specific)
(array, unequal) x 48,764,338 ops/sec $\pm 1.48\%$ (84 runs
sampled)

is-equal-shallow (object, equal) x 5,068,750 ops/sec
 $\pm 0.28\%$ (92 runs sampled) >is-equal-shallow (object, same)
x 17,231,997 ops/sec $\pm 0.42\%$ (92 runs sampled) >is-equal-
shallow (object, unequal) x 5,524,878 ops/sec $\pm 0.41\%$ (92
runs sampled) >is-equal-shallow (array, equal) x
1,067,063 ops/sec $\pm 0.40\%$ (92 runs sampled) >is-equal-
shallow (array, same) x 1,074,356 ops/sec $\pm 0.20\%$ (94 runs
sampled) >is-equal-shallow (array, unequal) x 1,758,859
ops/sec $\pm 0.44\%$ (92 runs sampled)

shallow-equals (object, equal) x 8,380,550 ops/sec $\pm 0.31\%$
(90 runs sampled) >shallow-equals (object, same) x
27,583,073 ops/sec $\pm 0.60\%$ (91 runs sampled) >shallow-
equals (object, unequal) x 8,954,268 ops/sec $\pm 0.71\%$ (92
runs sampled) >shallow-equals (array, equal) x
104,437,640 ops/sec $\pm 0.22\%$ (96 runs sampled) >shallow-
equals (array, same) x 141,850,542 ops/sec $\pm 0.25\%$ (93
runs sampled) >shallow-equals (array, unequal) x
47,964,211 ops/sec $\pm 1.51\%$ (84 runs sampled)

fbjs/lib/shallowEqual (object, equal) x 3,366,709 ops/sec
 $\pm 0.35\%$ (93 runs sampled) >fbjs/lib/shallowEqual (object,
same) x 794,825,194 ops/sec $\pm 0.24\%$ (94 runs sampled)
>fbjs/lib/shallowEqual (object, unequal) x 3,612,268 ops/
sec $\pm 0.37\%$ (94 runs sampled) >fbjs/lib/shallowEqual

```
(array, equal) x 1,613,800 ops/sec ±0.23% (90 runs sampled) >fbjs/lib/shallowEqual (array, same) x 794,861,384 ops/sec ±0.24% (93 runs sampled) >fbjs/lib/shallowEqual (array, unequal) x 1,648,398 ops/sec ±0.77% (92 runs sampled)
```

You can run the benchmarks yourselves by cloning the repository, installing dependencies, and running the `benchmark/index.js` script:

```
git clone https://github.com/WordPress/gutenberg.git
npm install
npm run build:packages
node ./packages/is-shallow-equal/benchmark
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/is-shallow-equal](#)

[Previous @wordpress/interface](#) [Previous: @wordpress/interface](#)
[Next @wordpress/jest-console](#) [Next: @wordpress/jest-console](#)

@wordpress/jest-console

In this article

[Table of Contents](#)

- [Installation](#)
 - [Setup](#)
 - [Usage](#)
 - [.toHaveErrored\(\)](#)
 - [.toHaveErroredWith\(arg1, arg2, ... \)](#)
 - [.toHaveInformed\(\)](#)
 - [.toHaveInformedWith\(arg1, arg2, ... \)](#)

- [.toHaveLogged\(\)](#)
- [.toHaveLoggedWith\(arg1, arg2, ... \)](#)
- [.toHaveWarned\(\)](#)
- [.toHaveWarnedWith\(arg1, arg2, ... \)](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Custom [Jest](#) matchers for the [Console](#) object to test JavaScript code in WordPress.

This package converts `console.error`, `console.info`, `console.log` and `console.warn` functions into mocks and tracks their calls.

It also enforces usage of one of the related matchers whenever tested code calls one of the mentioned `console` methods.

It means that you need to assert with `.toHaveErrored()` or `.toHaveErroredWith(arg1, arg2, ...)` when `console.error` gets executed, and use the corresponding methods when `console.info`, `console.log` or `console.warn` are called.

Your test will fail otherwise! This is a conscious design decision which helps to detect deprecation warnings when upgrading dependent libraries or smaller errors when refactoring code.

[Installation](#)

Install the module:

```
npm install @wordpress/jest-console --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[Setup](#)

The simplest setup is to use Jest's `setupFilesAfterEnv` config option:

```
"jest": {
  "setupFilesAfterEnv": [
    "@wordpress/jest-console"
  ]
},
```

[Usage](#)

[.toHaveErrored\(\)](#)

Use `.toHaveErrored` to ensure that `console.error` function was called.

For example, let's say you have a `drinkAll(flavor)` function that makes you drink all available beverages.

You might want to check if function calls `console.error` for 'octopus' instead, because 'octopus' flavor is really weird and why would anything be octopus-flavored? You can do that with this test suite:

```
describe( 'drinkAll', () => {
  test( 'drinks something lemon-flavored', () => {
    drinkAll( 'lemon' );
    expect( console ).not.toHaveErrored();
  } );
  test( 'errors when something is octopus-flavored', () => {
    drinkAll( 'octopus' );
    expect( console ).toHaveErrored();
  } );
});
```

.toHaveErroredWith(arg1, arg2, ...)

Use `.toHaveErroredWith` to ensure that `console.error` function was called with specific arguments.

For example, let's say you have a `drinkAll(flavor)` function again makes you drink all available beverages.

You might want to check if function calls `console.error` with a specific message for 'octopus' instead, because

'octopus' flavor is really weird and why would anything be octopus-flavored? To make sure this works, you could write:

```
describe( 'drinkAll', () => {
  test( 'errors with message when something is octopus-flavored', () => {
    drinkAll( 'octopus' );
    expect( console ).toHaveErroredWith(
      'Should I really drink something that is octopus-flavored?'
    );
  } );
});
```

.toHaveInformed()

Use `.toHaveInformed` to ensure that `console.info` function was called.

Almost identical usage as `.toHaveErrored()`.

.toHaveInformedWith(arg1, arg2, ...)

Use `.toHaveInformedWith` to ensure that `console.info` function was called with specific arguments.

Almost identical usage as `.toHaveErroredWith()`.

.toHaveLogged()

Use `.toHaveLogged` to ensure that `console.log` function was called.

Almost identical usage as `.toHaveErrored()`.

[.toHaveLoggedWith\(arg1, arg2, ... \)](#)

Use `.toHaveLoggedWith` to ensure that `console.log` function was called with specific arguments.

Almost identical usage as `.toHaveErroredWith()`.

[.toHaveWarned\(\)](#)

Use `.toHaveWarned` to ensure that `console.warn` function was called.

Almost identical usage as `.toHaveErrored()`.

[.toHaveWarnedWith\(arg1, arg2, ... \)](#)

Use `.toHaveWarnedWith` to ensure that `console.warn` function was called with specific arguments.

Almost identical usage as `.toHaveErroredWith()`.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/jest-console”](#)

[Previous @wordpress/is-shallow-equal](#) [Previous: @wordpress/is-shallow-equal](#)
[Next @wordpress/jest-preset-default](#) [Next: @wordpress/jest-preset-default](#)

@wordpress/jest-preset-default

In this article

Table of Contents

- [Installation](#)

- [Setup](#)
 - [Via jest.config.json or jest field in package.json](#)
 - [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Default [Jest](#) preset for WordPress development.

Installation

Install the module

```
npm install @wordpress/jest-preset-default --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

Setup

Via jest.config.json or jest field in package.json

```
{
  "preset": "@wordpress/jest-preset-default"
}
```

Usage

Brief explanations of options included

- `moduleNameMapper` – all `css` and `scss` files containing CSS styles will be stubbed out.
- `modulePaths` – the root dir of the project is used as a location to search when resolving modules.
- `setupFiles` – runs code before each test which sets up global variables required in the testing environment.
- `setupFilesAfterEnv` – runs code which adds improved support for `Console` object and `React` components to the testing framework before each test.
- `testEnvironment` – enabled the `jsdom` environment for all tests by default.
- `testMatch` – searches for tests in `/test/` and `/_tests_`/ subfolders, and also matches all files with a `.test.*` suffix. It detects test files with `.js`, `.jsx`, `.ts` or `.tsx` suffix. Compared to default Jest configuration, it doesn't match files with the `.spec.*` suffix.
- `testPathIgnorePatterns` – excludes `node_modules` and `vendor` directories from searching for test files.
- `transform` – keeps the default [babel-jest](#) transformer.

Using enzyme

Historically, this package used to use `enzyme`, but support was dropped in favor of `@testing-library/react`, primary reason being unblocking the upgrade to React 18.

If you wish to use `enzyme`, you can still use it by manually providing the React 17 adapter, by following the steps below.

To install the `enzyme` dependency, run:

```
npm install --save enzyme
```

To install the React 17 adapter dependency, run:

```
npm install --save @wojtekmaj/enzyme-adapter-react-17
```

To use the React 17 adapter, use this in your [setupFilesAfterEnv](#) configuration:

```
// It "mocks" enzyme, so that we can delay loading of
// the utility functions until enzyme is imported in tests.
// Props to @gdborton for sharing this technique in his article:
// https://medium.com/airbnb-engineering/unlocking-test-performance-migration-15333a2a2a2c
let mockEnzymeSetup = false;

jest.mock( 'enzyme', () => {
  const actualEnzyme = jest.requireActual( 'enzyme' );
  if ( ! mockEnzymeSetup ) {
    mockEnzymeSetup = true;

    // Configure enzyme 3 for React, from docs: http://airbnb.io/enzyme/docs/react.html#configuration
    const Adapter = jest.requireActual(
      '@wojtekmaj/enzyme-adapter-react-17'
    );
    actualEnzyme.configure( { adapter: new Adapter() } );
  }
  return actualEnzyme;
} );
```

If you also use snapshot tests with `enzyme`, you might want to add support for serializing them, through the `enzyme-to-json` package.

To install the dependency, run:

```
npm install --save enzyme-to-json
```

Finally, you should add `enzyme-to-json/serializer` to the array of [snapshotSerializers](#):

```
{
  snapshotSerializers: [ 'enzyme-to-json/serializer' ]
}
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/jest-preset-default”](#)

[Previous @wordpress/jest-console](#) [Previous: @wordpress/jest-console](#)

[Next @wordpress/library-export-default-webpack-plugin](#) [Next: @wordpress/library-export-default-webpack-plugin](#)

@wordpress/library-export-default-webpack-plugin

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

DEPRECATED for webpack v5: please use [output.library.export](#) instead.

Webpack plugin for exporting default property for selected libraries which use ES6 Modules. Implementation is based on the Webpack's core plugin [ExportPropertyMainTemplatePlugin](#). The only difference is that this plugin allows to include all entry point names where the default export of your entry point will be assigned to the library target.

[Installation](#)

Install the module

```
npm install @wordpress/library-export-default-webpack-plugin --save
```

Note: This package requires Node.js 12.0.0 or later. It is not compatible with older versions. It works only with webpack v4.

Usage

Construct an instance of `LibraryExportDefaultPlugin` in your Webpack configurations `plugins` entry, passing an array where values correspond to the entry point name.

The following example selects `boo` entry point to be updated by the plugin. When compiled, the built file will ensure that `default` value exported for the chunk will be assigned to the global variable `wp.boo`. `foo` chunk will remain untouched.

```
const LibraryExportDefaultPlugin = require( '@wordpress/library-export-default' )

module.exports = {
    // ...

    entry: {
        boo: './packages/boo',
        foo: './packages/foo',
    },
    output: {
        filename: 'build/[name].js',
        path: __dirname,
        library: [ 'wp', '[name]' ],
        libraryTarget: 'this',
    },
    plugins: [ new LibraryExportDefaultPlugin( [ 'boo' ] ) ],
};


```

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](#).

First published

March 9, 2021

Last updated

March 6, 2023

Edit article

[Improve it on GitHub: @wordpress/library-export-default-webpack-plugin](#)

[Previous: @wordpress/jest-preset-default](#) [Previous: @wordpress/jest-preset-default](#)
[Next: @wordpress/jest-puppeteer-axe](#) [Next: @wordpress/jest-puppeteer-axe](#)

@wordpress/jest-puppeteer-axe

In this article

Table of Contents

- [Installation](#)
 - [Setup](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

[Axe](#) (the Accessibility Engine) API integration with [Jest](#) and [Puppeteer](#).

Defines Jest async matcher to check whether a given Puppeteer's page instance passes [Axe](#) accessibility tests.

[Installation](#)

Install the module

```
npm install @wordpress/jest-puppeteer-axe --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[Setup](#)

The simplest setup is to use Jest's `setupFilesAfterEnv` config option:

```
"jest": {  
  "setupFilesAfterEnv": [  
    "@wordpress/jest-puppeteer-axe"  
  ]  
},
```

[Usage](#)

In your Jest test suite add the following code to the test's body:

```
test( 'checks the test page with Axe', async () => {  
  // First, run some code which loads the content of the page.  
  loadTestPage();  
  
  await expect( page ).toPassAxeTests();  
} );
```

It is also possible to pass optional params which allow Axe API to perform customized checks:

- `include` – CSS selector(s) to add the list of elements to include in analysis.

- `exclude` – CSS selector(s) to add the list of elements to exclude from analysis.
- `disabledRules` – the list of [Axe rules](#) to skip from verification.
- `options` – a flexible way to configure how Axe run operates. See [axe-core API documentation](#) for information on the object structure.
- `config` – Axe configuration object. See [axe-core API documentation](#) for documentation on the object structure.

```
test( 'checks the test component with Axe excluding some button', async () {
  // First, run some code which loads the content of the page.
  loadPageWithTestComponent();

  await expect( page ).toPassAxeTests( {
    include: '.test-component',
    exclude: '.some-button',
    disabledRules: [ 'aria-allowed-role' ],
    options: { iframes: false },
    config: { reporter: 'raw' },
  } );
} );
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/jest-puppeteer-axe”](#)

[Previous: @wordpress/library-export-default-webpack-plugin](#) [Previous: @wordpress/library-export-default-webpack-plugin](#)

[Next: @wordpress/keyboard-shortcuts](#) [Next: @wordpress/keyboard-shortcuts](#)

@wordpress/keyboard-shortcuts

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [ShortcutProvider](#)
 - [store](#)
 - [useShortcut](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Keyboard shortcuts is a generic package that allows registering and modifying shortcuts.

[Installation](#)

Install the module

```
npm install @wordpress/keyboard-shortcuts --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](#)

[ShortcutProvider](#)

Handles callbacks added to context by `useShortcut`. Adding a provider allows to register contextual shortcuts that are only active when a certain part of the UI is focused.

Parameters

- `props Object`: Props to pass to `div`.

Returns

- `Element`: Component.

[store](#)

Store definition for the keyboard shortcuts namespace.

Related

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

Type

- Object

[useShortcut](#)

Attach a keyboard shortcut handler.

Parameters

- *name* string: Shortcut name.
- *callback* Function: Shortcut callback.
- *options* Object: Shortcut options.
- *options.isDisabled* boolean: Whether to disable the shortcut.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/keyboard-shortcuts](#)

[Previous @wordpress/jest-puppeteer-axe](#) Previous: [@wordpress/jest-puppeteer-axe](#)
[Next @wordpress/keycodes](#) Next: [@wordpress/keycodes](#)

@wordpress/keycodes

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [API](#)
 - [ALT](#)
 - [BACKSPACE](#)
 - [COMMAND](#)

- [CTRL](#)
- [DELETE](#)
- [displayShortcut](#)
- [displayShortcutList](#)
- [DOWN](#)
- [END](#)
- [ENTER](#)
- [ESCAPE](#)
- [F10](#)
- [HOME](#)
- [isAppleOS](#)
- [isKeyboardEvent](#)
- [LEFT](#)
- [modifiers](#)
- [PAGEDOWN](#)
- [PAGEUP](#)
- [rawShortcut](#)
- [RIGHT](#)
- [SHIFT](#)
- [shortcutAriaLabel](#)
- [SPACE](#)
- [TAB](#)
- [UP](#)
- [ZERO](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Keycodes utilities for WordPress, used to check the key pressed in events like `onKeyDown`. Contains keycodes constants for keyboard keys like `DOWN`, `UP`, `ENTER`, etc.

Installation

Install the module

```
npm install @wordpress/keycodes --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.

Usage

Check which key was used in an `onKeyDown` event:

```
import { DOWN, ENTER } from '@wordpress/keycodes';

// [...]

onKeyDown( event ) {
    const { keyCode } = event;
```

```
if ( keyCode === DOWN ) {
    alert( 'You pressed the down arrow!' );
} else if ( keyCode === ENTER ) {
    alert( 'You pressed the enter key!' );
} else {
    alert( 'You pressed another key.' );
}
}
```

[API](#)

[ALT](#)

Keycode for ALT key.

[BACKSPACE](#)

Keycode for BACKSPACE key.

[COMMAND](#)

Keycode for COMMAND/META key.

[CTRL](#)

Keycode for CTRL key.

[DELETE](#)

Keycode for DELETE key.

[displayShortcut](#)

An object that contains functions to display shortcuts.

Usage

```
// Assuming macOS:  
displayShortcut.primary( 'm' );  
// "⌘M"
```

Type

- `WPMODIFIERHANDLER<WPKEYHANDLER<STRING>>`Keyed map of functions to display shortcuts.

[displayShortcutList](#)

Return an array of the parts of a keyboard shortcut chord for display.

Usage

```
// Assuming macOS:  
displayShortcutList.primary( 'm' );  
// [ "⌘", "M" ]
```

Type

- `WPModifierHandler<WPKeyHandler<string[]>>` Keyed map of functions to shortcut sequences.

[DOWN](#)

Keycode for DOWN key.

[END](#)

Keycode for END key.

[ENTER](#)

Keycode for ENTER key.

[ESCAPE](#)

Keycode for ESCAPE key.

[F10](#)

Keycode for F10 key.

[HOME](#)

Keycode for HOME key.

[isAppleOS](#)

Return true if platform is MacOS.

Parameters

- `_window Window?`: window object by default; used for DI testing.

Returns

- `boolean`: True if MacOS; false otherwise.

[isKeyboardEvent](#)

An object that contains functions to check if a keyboard event matches a predefined shortcut combination.

Usage

```
// Assuming an event for ⌘M key press:  
isKeyboardEvent.primary( event, 'm' );  
// true
```

Type

- `WPMODIFIERHANDLER<WPKEYHANDLER>`Keyed map of functions to match events.

LEFT

Keycode for LEFT key.

modifiers

Object that contains functions that return the available modifier depending on platform.

Type

- `WPMODIFIERHANDLER< (isApple: () => boolean) => WPMODIFIERPART[]>`

PAGEDOWN

Keycode for PAGEDOWN key.

PAGEUP

Keycode for PAGEUP key.

rawShortcut

An object that contains functions to get raw shortcuts.

These are intended for user with the KeyboardShortcuts.

Usage

```
// Assuming macOS:  
rawShortcut.primary( 'm' );  
// "meta+m"
```

Type

- `WPMODIFIERHANDLER<WPKEYHANDLER<string>>`Keyed map of functions to raw shortcuts.

RIGHT

Keycode for RIGHT key.

[SHIFT](#)

Keycode for SHIFT key.

[shortcutAriaLabel](#)

An object that contains functions to return an aria label for a keyboard shortcut.

Usage

```
// Assuming macOS:  
shortcutAriaLabel.primary( '.' );  
// "Command + Period"
```

Type

- `WPModifierHandler<WPKeyHandler<string>>`Keyed map of functions to shortcut ARIA labels.

[SPACE](#)

Keycode for SPACE key.

[TAB](#)

Keycode for TAB key.

[UP](#)

Keycode for UP key.

[ZERO](#)

Keycode for ZERO key.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/keycodes”](#)

[Previous @wordpress/keyboard-shortcuts](#) [Previous: @wordpress/keyboard-shortcuts](#)
[Next @wordpress/lazy-import](#) [Next: @wordpress/lazy-import](#)

@wordpress/lazy-import

In this article

[Table of Contents](#)

- [Installation](#)
- [Requirements](#)
- [Usage](#)
 - [Options](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Import an NPM module, even if not installed locally or defined as a dependency of the project. Uses a locally installed package if available. Otherwise, the package will be downloaded dynamically on-demand.

[Installation](#)

Install the module

```
npm install @wordpress/lazy-import --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.*

[Requirements](#)

NPM 6.9.0 or newer is required, since it uses the [package aliases feature](#) to maintain multiple versions of the same package.

[Usage](#)

Usage is intended to mimic the behavior of the [dynamic import function](#), receiving the name (and optional version specifier) of an NPM package and returning a promise which resolves to the required module.

Note: Currently, this alignment to `import` is superficial, and the module resolution still uses [CommonJS require](#), rather than the newer [ES Modules support](#). Future versions of this

package will likely support ES Modules, once an LTS release of Node.js including unflagged ES Modules support becomes available.

The string passed to `lazyImport` can be formatted exactly as you would provide to `npm install`, including an optional version specifier (including [version ranges](#)). If the version specifier is omitted, it will be treated as equivalent to `*`, using the version of a locally installed package if available, otherwise installing the latest available version.

```
const lazyImport = require( '@wordpress/lazy-import' );

lazyImport( 'is-equal-shallow@^0.1.3' ).then( ( isEqualShallow ) => {
    console.log( isEqualShallow( { a: true, b: true }, { a: true, b: true } ) );
} );
```

If you're using Node v14.3.0 or newer, you can also take advantage of [top-level await](#) to simplify top-level imports:

```
const lazyImport = require( '@wordpress/lazy-import' );

const isEqualShallow = await lazyImport( 'is-equal-shallow@^0.1.3' );
console.log( isEqualShallow( { a: true, b: true }, { a: true, b: true } ) );
// true
```

`lazyImport` optionally accepts a second argument, an options object:

```
const lazyImport = require( '@wordpress/lazy-import' );

function onInstall() {
    console.log( 'Installing...' );
}

lazyImport( 'fbjs@^1.0.0', {
    localPath: './lib/shallowEqual',
    onInstall,
} ).then( /* ... */ );
```

Note that `lazyImport` can throw an error when offline and unable to install the dependency using NPM. You may want to anticipate this and provide remediation steps for a failed install, such as logging a warning message:

```
try {
    await lazyImport( 'is-equal-shallow@^0.1.3' );
} catch ( error ) {
    if ( error.code === 'ENOTFOUND' ) {
        console.log( 'Unable to connect to NPM registry!' );
    }
}
```

[Options](#)

`localPath`

- Type: `string`

- Required: No

Local path pointing to a file or directory that can be used when other script that `main` needs to be imported.

`onInstall`

- Type: `Function`
- Required: No

Function to call if and when the module is being installed. Since installation can cause a delay in script execution, this can be useful to output logging information or display a spinner.

An installation can be assumed to finish once the returned promise is resolved.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/lazy-import”](#)

[Previous @wordpress/keycodes](#) Previous: [@wordpress/keycodes](#)

Next [@wordpress/react-native-aztec](#) Next: [@wordpress/react-native-aztec](#)

@wordpress/react-native-aztec

In this article

Table of Contents

- [RCTAztecView](#)
 - [Usage](#)
 - [Props](#)
 - [text](#)
 - [blockType](#)
 - [isMultiline](#)

- [activeFormats](#)
- [disableEditingMenu](#)
- [maxImagesWidth](#)
- [minWidth](#)
- [maxWidth](#)
- [fontFamily](#)
- [fontSize](#)
- [fontWeight](#)
- [fontStyle](#)
- [deleteEnter](#)
- [color](#)
- [selectionColor](#)
- [placeholder](#)
- [placeholderTextColor](#)
- [textAlign](#)
- [onChange\(value: Event \)](#)
- [onKeyDown\(value: Event \)](#)
- [onFocus\(value: Event \)](#)
- [onBlur\(value: Event \)](#)
- [onPaste\(value: Event \)](#)
- [onContentSizeChange\(value: Event \)](#)
- [onCaretVerticalPositionChange\(value: Event \)](#)
- [onSelectionChange\(value: Event \)](#)
- [Native Implementation details](#)
 - [iOS](#)
 - [Android](#)
- [License](#)
 - [Contributing to this package](#)

[↑ Back to top](#)

This package provides a component, AztecView, that wraps around the Aztec Android and Aztec iOS libraries in a React Native component.

This component provides rich text editing capabilities that emulate a subset of the HTML functionality.

RCTAztecView

Render a rich text area that displays the HTML content provided.

Usage

```
import RCTAztecView from '@wordpress/react-native-aztec';

const RichText = () => (
  <>
    <RCTAztecView
      text={ {
        text: '<h1>This is a Heading</h1>',
        selection: { start: 0, end: 0 },
      } }>
```

```
    />
  </>
);
```

Props

text

Object with current HTML string to make editable and selection/caret position.

- Type: Object, with the following attributes:
 - text: HTML content
 - selection:
 - start, start position of selection
 - end: end position of selection
 - eventCount: if it has a value it's because this change was originated from the native event.
- Required: Yes

blockType

The block type, should contain a tagName prop that indicates what is the HTML tag that this editor displays.

- Type: Object
- Required: No
- Android only

isMultiline

By default, a line break will be inserted on Enter. If the editable field can contain multiple paragraphs, this property can be set to create new paragraphs on Enter.

- Type: Boolean
- Required: No

activeFormats

The formats that are currently active. This is reflected on current state of the cursor.

- Type: Array
- Required: No

disableEditingMenu

If active disables the contextual menu that allows setting text attributes like Bold/Italic/Strikethrough.

- Type: Boolean
- Required: No

maxImagesWidth

The maximum width an image that is part of content provided can have.

- Type: Number
- Required: No

minWidth

The minimum width the component can have.

- Type: Number
- Required: No

maxWidth

The maximum width the component can have.

- Type: Number
- Required: No

fontFamily

The font family that will be used as default to display the HTML content.

- Type: String
- Required: No

fontSize

The font size that will be used as default to display the HTML content.

- Type: Number
- Required: No

fontWeight

The font weight that will be used as default to display the HTML content.

- Type: String
- Required: No

fontStyle

The font style (bold, italic,) that will be used as default to display the HTML content.

- Type: String
- Required: No

[deleteEnter](#)

When active removes the new line resulting from an enter keypress when that enter keypress is splitting the block.

- Type: Boolean
- Required: No
- Android Only

[color](#)

Text color.

- Type: Color
- Required: No

[selectionColor](#)

The color to use for the caret and for the selection background.

- Type: Color
- Required: No

[placeholder](#)

Placeholder text to show when the field is empty.

- Type: String
- Required: No

[placeholderTextColor](#)

Placeholder text color.

- Type: Color
- Required: No

[textAlign](#)

The alignment for the text displayed. Possible values: Left, Right, Center, Justify.

- Type: String
- Required: No

[onChange\(value: Event \)](#)

- Type: function
- Required: No

onKeyDown(value: Event)

Called when a key that belongs the triggerKeyCodes props is pressed.

- Type: `function`
- Required: No

onFocus(value: Event)

Called when then native component is focused on, for example when tapped.

- Type: `function`
- Required: No

onBlur(value: Event)

Called when then native component lost the focus.

- Type: `function`
- Required: No

onPaste(value: Event)

Called when then native component has content pasted in.

- Type: `function`
- Required: No

onContentSizeChange(value: Event)

Called when then native component size changed.

- Type: `function`
- Required: No

onCaretVerticalPositionChange(value: Event)

Called when the vertical position of the caret changed. This can be used to scroll the container of the component to keep the caret in focus.

- Type: `function`
- Required: No

onSelectionChange(value: Event)

Called when then selection of the native component changed.

- Type: `function`
- Required: No

Native Implementation details

iOS

On iOS we use a native view called RCTAztecView that inherits an Aztec TextView class. RCTAztecView adds the following custom behaviours to the TextView class:

- Overlays a UILabel to display placeholder text
- Overrides the `onPaste` method to intercept paste actions and send them to the JS implementation
- Overrides the `insertText` and `deleteBackward` methods in order to detect the following keypresses:
 - delete/backspace to allow handling of custom merge actions
 - enter/new lines to allow handling of custom split actions
 - detection any of triggerKeyCodes
- Sets the `characterToReplaceLastEmptyLine` property in the HTMLConverter to be zero width space character to avoid the insertion of a newline at the end of the text blocks
- Disables the `shouldCollapseSpaces` flag in the HTMLConverter in order to maintain all spaces inserted by the user

Android

Android uses a native [ReactAztecText](#) view, which extends [AztecText](#) from the [Aztec Library for Android](#). All interactions between the native ReactAztecText view and the JavaScript code are handled by the [ReactAztecManager](#) view manager.

License

GPL v2

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](#).

First published

March 9, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: @wordpress/react-native-aztec”](#)

[Previous @wordpress/lazy-import](#) [Previous: @wordpress/lazy-import](#)

[Next @wordpress/list-reusable-blocks](#) [Next: @wordpress/list-reusable-blocks](#)

@wordpress/list-reusable-blocks

In this article

[Table of Contents](#)

- [Installation](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Package used to add import/export links to the listing page of the reusable blocks.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

Installation

Install the module

```
npm install @wordpress/list-reusable-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.*

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/list-reusable-blocks”](#)

[Previous @wordpress/react-native-aztec](#) [Previous: @wordpress/react-native-aztec](#)
[Next @wordpress/react-native-bridge](#) [Next: @wordpress/react-native-bridge](#)

@wordpress/react-native-bridge

In this article

[Table of Contents](#)

- [Getting started](#)
 - [Mostly automatic installation](#)
 - [Manual installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

[Getting started](#)

This package is not yet published to npm. You can use it locally:

```
$ npm install ./gutenberg/packages/react-native-bridge --save
```

[Mostly automatic installation](#)

```
$ react-native link @wordpress/react-native-bridge
```

[Manual installation](#)

iOS

1. In Xcode, in the project navigator, right click **Libraries** ➔ Add Files to [your project's name]
2. Go to `node_modules` ➔ `react-native-bridge` and add `RNReactNativeGutenbergBridge.xcodeproj`
3. In Xcode, in the project navigator, select your project. Add `libRNReactNativeGutenbergBridge.a` to your project's Build Phases ➔ Link Binary With Libraries
4. Run your project (Cmd+R)<

Android

1. Open up android/app/src/main/java/[...]/MainActivity.java
 - Add import
com.reactlibrary.RNReactNativeGutenbergBridgePackage; to the imports at the top of the file
 - Add new RNReactNativeGutenbergBridgePackage() to the list returned by the getPackages() method
1. Append the following lines to android/settings.gradle:

```
include ':@wordpress_react-native-bridge'  
project(':@wordpress_react-native-bridge').projectDir = new  
File(rootProject.projectDir, './gutenberg/packages/react-  
native-bridge/android')
```
2. Insert the following lines inside the dependencies block in android/app/build.gradle:

```
implementation project(':@wordpress_react-native-bridge')
```

Usage

```
import RNReactNativeGutenbergBridge from '@wordpress/react-native-bridge';  
  
// TODO: What to do with the module?  
RNReactNativeGutenbergBridge;
```

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](#).

First published

March 9, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: @wordpress/react-native-bridge](#)

[Previous: @wordpress/list-reusable-blocks](#) [Previous: @wordpress/list-reusable-blocks](#)
[Next: @wordpress/media-utils](#) [Next: @wordpress/media-utils](#)

@wordpress/media-utils

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
 - [uploadMedia](#)
 - [MediaUpload](#)
- [Contributing to this package](#)

[↑ Back to top](#)

The media utils package provides a set of artifacts to abstract media functionality that may be useful in situations where there is a need to deal with media uploads or with the media library, e.g., artifacts that extend or implement a block-editor.

This package is meant to be used by the WordPress core. It may not work as expected outside WordPress usages.

[Installation](#)

Install the module

```
npm install @wordpress/media-utils --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.*

[Usage](#)

[uploadMedia](#)

Media upload util is a function that allows the invokers to upload files to the WordPress media library.

As an example, provided that `myFiles` is an array of file objects, `handleFileChange` on `onFileChange` is a function that receives an array of objects containing the description of WordPress media items and `handleFileError` is a function that receives an object describing a possible error, the following code uploads a file to the WordPress media library:

```
wp.mediaUtils.utils.uploadMedia( {  
    fileList: myFiles,  
    onChange: handleFileChange,  
    onError: handleFileError,  
} );
```

The following code uploads a file named `foo.txt` with `foo` as content to the media library and alerts its URL:

```
wp.mediaUtils.utils.uploadMedia( {
  fileList: [ new File( [ 'foo' ], 'foo.txt', { type: 'text/plain' } ) ],
  onChange: ( [ fileObj ] ) => alert( fileObj.url ),
  onError: console.error,
} );
```

Beware that first onChange is called with temporary blob URLs and then with the final URL's this allows to show the result in an optimistic UI as if the upload was already completed. E.g.: when uploading an image, one can show the image right away in the UI even before the upload is complete.

[MediaUpload](#)

Media upload component provides a UI button that allows users to open the WordPress media library. It is normally used in conjunction with the filter `editor.MediaUpload`. The component follows the interface specified in <https://github.com/WordPress/gutenberg/blob/HEAD/packages/block-editor/src/components/media-upload/README.md>, and more details regarding its usage can be checked there.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/media-utils”](#)

[Previous @wordpress/react-native-bridge](#) [Previous: @wordpress/react-native-bridge](#)
[Next @wordpress/react-native-editor](#) [Next: @wordpress/react-native-editor](#)

@wordpress/react-native-editor

In this article

[Table of Contents](#)

- [Getting Started](#)
- [License](#)

- [Contributing to this package](#)

[↑ Back to top](#)

This package provides a demo application to simplify the environment setup required for the development of Gutenberg for native Android and iOS. The demo application allows running the mobile versions of Gutenberg blocks while avoiding the additional setup steps required by the [WordPress Android](#) and [WordPress iOS](#) apps.

Getting Started

Please review [Getting Started for the React Native based Mobile Gutenberg](#) to learn how to set up and run this demo application.

License

Gutenberg Mobile is an Open Source project covered by the [GNU General Public License version 2](#).

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](#).

First published

March 9, 2021

Last updated

January 16, 2024

Edit article

[Improve it on GitHub: @wordpress/react-native-editor”](#)

[Previous](#) [@wordpress/media-utils](#) [Previous: @wordpress/media-utils](#)
[Next](#) [@wordpress/notices](#) [Next: @wordpress/notices](#)

@wordpress/notices

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

State management for notices.

[Installation](#)

Install the module

```
npm install @wordpress/notices
```

*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.*

[Usage](#)

When imported, the notices module registers a data store on the `core/notices` namespace. In WordPress, this is accessed from `wp.data.dispatch('core/notices')`.

For more information about consuming from a data store, refer to [the @wordpress/data documentation on Data Access and Manipulation](#).

For a full list of actions and selectors available in the `core/notices` namespace, refer to the [Notices Data Handbook page](#).

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: @wordpress/notices”](#)

[Previous @wordpress/react-native-editor](#) [Previous: @wordpress/react-native-editor](#)

[Next @wordpress/npm-package-json-lint-config](#) [Next: @wordpress/npm-package-json-lint-config](#)

@wordpress/npm-package-json-lint-config

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

WordPress [npm-package-json-lint](#) shareable configuration.

[Installation](#)

Install the module

```
$ npm install @wordpress/npm-package-json-lint-config
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[Usage](#)

Add this to your `package.json` file:

```
"npmpackagejsonlint": {  
    "extends": "@wordpress/npm-package-json-lint-config",  
},
```

Or to a `.npmpackagejsonlintrc.json` file in the root of your repo:

```
{  
    "extends": "@wordpress/npm-package-json-lint-config"  
}
```

To add, modify, or override any [npm-package-json-lint](#) rules add this to your `package.json` file:

```
"npmpackagejsonlint": {  
    "extends": "@wordpress/npm-package-json-lint-config",  
    "rules": {  
        "valid-values-author": [  
            "error",  
            [  
                "WordPress"  
            ]  
        ]  
    }  
},
```

Or to a `.npmpackagejsonlintrc.json` file in the root of your repo:

```
{  
    "extends": "@wordpress/npm-package-json-lint-config",  
    "rules": {  
        "require-publishConfig": "error",  
        "valid-values-author": [ "error", [ "WordPress" ] ]  
    }  
}
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/npm-package-json-lint-config](#)

[Previous](#) [@wordpress/notices](#) [Previous: @wordpress/notices](#)

[Next](#) [@wordpress/nux](#) [Next: @wordpress/nux](#)

@wordpress/nux

In this article

Table of Contents

- [Installation](#)
- [DotTip](#)
- [Determining if a tip is visible](#)
- [Manually dismissing a tip](#)
- [Disabling and enabling tips](#)
- [Triggering a guide](#)
- [Getting information about a guide](#)
- [Contributing to this package](#)

[↑ Back to top](#)

The NUX module exposes components, and `wp.data` methods useful for onboarding a new user to the WordPress admin interface. Specifically, it exposes *tips* and *guides*.

A *tip* is a component that points to an element in the UI and contains text that explains the element's functionality. The user can dismiss a tip, in which case it never shows again. The user can also disable tips entirely. Information about tips is persisted between sessions using `localStorage`.

A *guide* allows a series of tips to be presented to the user one by one. When a user dismisses a tip that is in a guide, the next tip in the guide is shown.

[Installation](#)

Install the module

```
npm install @wordpress/nux --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.*

[DotTip](#)

`DotTip` is a React component that renders a single *tip* on the screen. The tip will point to the React element that `DotTip` is nested within. Each tip is uniquely identified by a string passed to `tipId`.

See [the component's README](#) for more information.

```
<button onClick={ ... }>
  Add to Cart
  <DotTip tipId="acme/add-to-cart">
    Click here to add the product to your shopping cart.
  </DotTip>
</button>
```

```
</DotTip>
</button>
}
```

Determining if a tip is visible

You can programmatically determine if a tip is visible using the `isTipVisible` select method.

```
const isVisible = select( 'core/nux' ).isTipVisible( 'acme/add-to-cart' );
console.log( isVisible ); // true or false
```

Manually dismissing a tip

`dismissTip` is a dispatch method that allows you to programmatically dismiss a tip.

```
<button
  onClick={ () => {
    dispatch( 'core/nux' ).dismissTip( 'acme/add-to-cart' );
  } }
>
  Dismiss tip
</button>
```

Disabling and enabling tips

Tips can be programmatically disabled or enabled using the `disableTips` and `enableTips` dispatch methods. You can query the current setting by using the `areTipsEnabled` select method.

Calling `enableTips` will also un-dismiss all previously dismissed tips.

```
const areTipsEnabled = select( 'core/nux' ).areTipsEnabled();
return (
  <button
    onClick={ () => {
      if ( areTipsEnabled ) {
        dispatch( 'core/nux' ).disableTips();
      } else {
        dispatch( 'core/nux' ).enableTips();
      }
    } }
  >
    { areTipsEnabled ? 'Disable tips' : 'Enable tips' }
  </button>
);
```

Triggering a guide

You can group a series of tips into a guide by calling the `triggerGuide` dispatch method. The given tips will then appear one by one.

A tip cannot be added to more than one guide.

```
dispatch( 'core/nux' ).triggerGuide( [  
    'acme/product-info',  
    'acme/add-to-cart',  
    'acme/checkout',  
] );
```

[Getting information about a guide](#)

`getAssociatedGuide` is a select method that returns useful information about the state of the guide that a tip is associated with.

```
const guide = select( 'core/nux' ).getAssociatedGuide( 'acme/add-to-cart' );  
console.log( 'Tips in this guide:', guide.tipIds );  
console.log( 'Currently showing:', guide.currentTipId );  
console.log( 'Next to show:', guide.nextTipId );
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/nux”](#)

[Previous: @wordpress/npm-package-json-lint-config](#) [Previous: @wordpress/npm-package-json-lint-config](#)

[Next: @wordpress/patterns](#) [Next: @wordpress/patterns](#)

@wordpress/patterns

In this article

Table of Contents

- [Installation](#)

- [Components](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Note

This package is currently only used internally by the Gutenberg project to manage the creation and editing of user patterns using the `wp_block` CPT in the context of the block editor. The likes of the `PatternsMenuItems` component expect to be rendered within a `BlockEditorProvider` in order to work.

Installation

Install the module

```
npm install @wordpress/patterns --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.

Components

This package doesn't currently have any publically exported components.

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](#).

First published

August 15, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: `@wordpress/patterns`](#)

[Previous: `@wordpress/nux`](#) [Previous: `@wordpress/nux`](#)
[Next: `@wordpress/plugins`](#) [Next: `@wordpress/plugins`](#)

@wordpress/plugins

In this article

Table of Contents

- [Installation](#)
 - [Plugins API](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Plugins module for WordPress.

[Installation](#)

Install the module

```
npm install @wordpress/plugins --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.*

[Plugins API](#)

getPlugin

Returns a registered plugin settings.

Parameters

- `name` `string`: Plugin name.

Returns

- `WPPlugin` | `undefined`: Plugin setting.

getPlugins

Returns all registered plugins without a scope or for a given scope.

Parameters

- `scope` `string`: The scope to be used when rendering inside a plugin area. No scope by default.

Returns

- `WPPlugin[]`: The list of plugins without a scope or for a given scope.

PluginArea

A component that renders all plugin fills in a hidden div.

Usage

```
// Using ES5 syntax
var el = React.createElement;
var PluginArea = wp.plugins.PluginArea;

function Layout() {
    return el( 'div', { scope: 'my-page' }, 'Content of the page', PluginArea );
}

// Using ESNext syntax
import { PluginArea } from '@wordpress/plugins';

const Layout = () => (
    <div>
        Content of the page
        <PluginArea scope="my-page" />
    </div>
);


```

Parameters

- `props { scope?: string; onError?: (name: WPPlugin['name'], error: Error) => void; }:`
- `props.scope` string:
- `props.onError (name: WPPlugin['name'], error: Error) => void:`

Returns

- Component: The component to be rendered.

registerPlugin

Registers a plugin to the editor.

Usage

```
// Using ES5 syntax
var el = React.createElement;
var Fragment = wp.element.Fragment;
var PluginSidebar = wp.editPost.PluginSidebar;
var PluginSidebarMoreMenuItem = wp.editPost.PluginSidebarMoreMenuItem;
var registerPlugin = wp.plugins.registerPlugin;
var moreIcon = React.createElement( 'svg' ); //... svg element.

function Component() {
    return el(
        Fragment,
        {},
        el(
```

```

        PluginSidebarMoreMenuItem,
        {
            target: 'sidebar-name',
        },
        'My Sidebar'
    ),
    el(
        PluginSidebar,
        {
            name: 'sidebar-name',
            title: 'My Sidebar',
        },
        'Content of the sidebar'
    )
);
}
registerPlugin( 'plugin-name' , {
    icon: moreIcon,
    render: Component,
    scope: 'my-page',
} );
// Using ESNext syntax
import { PluginSidebar, PluginSidebarMoreMenuItem } from '@wordpress/edit-
import { registerPlugin } from '@wordpress/plugins';
import { more } from '@wordpress/icons';

const Component = () => (
    <>
        <PluginSidebarMoreMenuItem target="sidebar-name">
            My Sidebar
        </PluginSidebarMoreMenuItem>
        <PluginSidebar name="sidebar-name" title="My Sidebar">
            Content of the sidebar
        </PluginSidebar>
    </>
);
registerPlugin( 'plugin-name' , {
    icon: more,
    render: Component,
    scope: 'my-page',
} );

```

Parameters

- ***name* string:** A string identifying the plugin. Must be unique across all registered plugins.
- ***settings* PluginSettings:** The settings for this plugin.

Returns

- **PluginSettings | null:** The final plugin settings object.

unregisterPlugin

Unregisters a plugin by name.

Usage

```
// Using ES5 syntax
var unregisterPlugin = wp.plugins.unregisterPlugin;

unregisterPlugin( 'plugin-name' );

// Using ESNext syntax
import { unregisterPlugin } from '@wordpress/plugins';

unregisterPlugin( 'plugin-name' );
```

Parameters

- *name* `string`: Plugin name.

Returns

- `WPPlugin | undefined`: The previous plugin settings object, if it has been successfully unregistered; otherwise `undefined`.

usePluginContext

A hook that returns the plugin context.

Returns

- `PluginContext`: Plugin context

withPluginContext

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

Parameters

- `mapContextToProps (context: PluginContext, props: T) => T & PluginContext`: Function called on every context change, expected to return object of props to merge with the component's own props.

Returns

- `Component`: Enhanced component with injected context as props.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/plugins”](#)

[Previous @wordpress/patterns](#) [Previous: @wordpress/patterns](#)

[Next @wordpress/postcss-plugins-preset](#) [Next: @wordpress/postcss-plugins-preset](#)

@wordpress/postcss-plugins-preset

In this article

[Table of Contents](#)

- [Installation](#)
- [Contributing to this package](#)

[↑ Back to top](#)

PostCSS sharable plugins preset for WordPress development.

[Installation](#)

Install the module

```
npm install @wordpress/postcss-plugins-preset --save
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/postcss-plugins-preset”](#)

[Previous @wordpress/plugins](#) [Previous: @wordpress/plugins](#)
[Next @wordpress/postcss-themes](#) [Next: @wordpress/postcss-themes](#)

@wordpress/postcss-themes

In this article

[Table of Contents](#)

- [Installation](#)
- [Contributing to this package](#)

[↑ Back to top](#)

PostCSS plugin to generate theme colors.

[Installation](#)

Install the module

```
npm install @wordpress/postcss-themes --save
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/postcss-themes”](#)

[Previous @wordpress/postcss-plugins-preset](#) [Previous: @wordpress/postcss-plugins-preset](#)
[Next @wordpress/preferences-persistence](#) [Next: @wordpress/preferences-persistence](#)

@wordpress/preferences-persistence

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
- [Reference](#)
 - [create](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Persistence utilities for `wordpress/preferences`.

Includes a persistence layer that saves data to WordPress user meta via the REST API. If for any reason data cannot be saved to the database, this persistence layer also uses local storage as a fallback.

[Installation](#)

Install the module

```
npm install @wordpress/preferences-persistence --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.*

[Usage](#)

Call the `create` function to create a persistence layer.

```
const persistenceLayer = create();
```

Next, configure the preferences package to use this persistence layer:

```
wp.data( 'core/preferences' ).setPersistenceLayer( persistenceLayer );
```

[Reference](#)

[create](#)

Creates a persistence layer that stores data in WordPress user meta via the REST API.

Parameters

- *options Object*:
- *options.preloadedData ?Object*: Any persisted preferences data that should be preloaded. When set, the persistence layer will avoid fetching data from the REST API.
- *options.localStorageRestoreKey ?string*: The key to use for restoring the localStorage backup, used when the persistence layer calls `localStorage.getItem` or `localStorage.setItem`.
- *options.requestDebounceMS ?number*: Debounce requests to the API so that they only occur at minimum every `requestDebounceMS` milliseconds, and don't swamp the server. Defaults to 2500ms.

Returns

- `Object`: A persistence layer for WordPress user meta.

[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](#).

First published

April 27, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/preferences-persistence”](#)

[Previous @wordpress/postcss-themes](#) Previous: [@wordpress/postcss-themes](#)
[Next @wordpress/preferences](#) Next: [@wordpress/preferences](#)

@wordpress/preferences

In this article

Table of Contents

- [Installation](#)

- [Key concepts](#)
 - [Scope](#)
 - [Key](#)
 - [Value](#)
 - [Defaults](#)
- [Examples](#)
 - [Data store](#)
 - [Components](#)
- [API Reference](#)
 - [Actions](#)
 - [Selectors](#)
- [Contributing to this package](#)

[↑ Back to top](#)

A key/value store for application preferences.

Installation

Install the module

```
npm install @wordpress/preferences --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.*

Key concepts

Scope

Many API calls require a ‘scope’ parameter that acts like a namespace. If you have multiple parameters with the same key but they apply to different parts of your application, using scopes is the best way to segregate them.

Key

Each preference is set against a key that should be a string.

Value

Values can be of any type, but the types supported may be limited by the persistence layer configure. For example if preferences are saved to browser localStorage in JSON format, only JSON serializable types should be used.

Defaults

Defaults are the value returned when a preference is `undefined`. These are not persisted, they are only kept in memory. They should be during the initialization of an application.

Examples

Data store

Set the default preferences for any features on initialization by dispatching an action:

```
import { dispatch } from '@wordpress/data';
import { store as preferencesStore } from '@wordpress/preferences';

function initialize() {
    // ...

    dispatch( preferencesStore ).setDefaults(
        'namespace/editor-or-plugin-name',
        {
            myBooleanFeature: true,
        }
    );
}

// ...
}
```

Use the `get` selector to get a preference value, and the `set` action to update a preference:

```
wp.data
    .select( 'core/preferences' )
    .get( 'namespace/editor-or-plugin-name', 'myPreferenceName' ); // 1
wp.data
    .dispatch( 'core/preferences' )
    .set( 'namespace/editor-or-plugin-name', 'myPreferenceName', 2 );
wp.data
    .select( 'core/preferences' )
    .get( 'namespace/editor-or-plugin-name', 'myPreferenceName' ); // 2
```

Use the `toggle` action to flip a boolean preference between `true` and `false`:

```
wp.data
    .select( 'core/preferences' )
    .get( 'namespace/editor-or-plugin-name', 'myPreferenceName' ); // true
wp.data
    .dispatch( 'core/preferences' )
    .toggle( 'namespace/editor-or-plugin-name', 'myPreferenceName' );
wp.data
    .select( 'core/preferences' )
    .get( 'namespace/editor-or-plugin-name', 'myPreferenceName' ); // false
```

Setting up a persistence layer

By default, this package only stores values in-memory. But it can be configured to persist preferences to browser storage or a database via an optional persistence layer.

Use the `setPersistenceLayer` action to configure how the store persists its preference values.

```

wp.data.dispatch( 'core/preferences' ).setPersistenceLayer( {
    // `get` is asynchronous to support persisting preferences using a REST API.
    // It will immediately be called by `setPersistenceLayer` and the returned value used as the initial state of the preferences.
    async get() {
        return JSON.parse( window.localStorage.getItem( 'MY_PREFERENCES' ) )
    },
    // `set` is synchronous. It's ok to use asynchronous code, but the preferences store won't wait for a promise to resolve, the function is 'fire and forget'.
    set( preferences ) {
        window.localStorage.setItem(
            'MY_PREFERENCES',
            JSON.stringify( preferences )
        );
    },
} );

```

For application that persist data to an asynchronous API, a concern is that loading preferences can lead to slower application start up.

A recommendation is to pre-load any persistence layer data and keep it in a local cache particularly if you're using an asynchronous API to persist data.

Note: currently `get` is called only when `setPersistenceLayer` is triggered. This may change in the future, so it's sensible to optimize `get` using a local cache, as shown in the example below.

```

// Preloaded data from the server.
let cache = preloadedData;
wp.data.dispatch( 'core/preferences' ).setPersistenceLayer( {
    async get() {
        if ( cache ) {
            return cache;
        }

        // Call to a made-up async API.
        return await api.preferences.get();
    },
    set( preferences ) {
        cache = preferences;
        api.preferences.set( { data: preferences } );
    },
} );

```

Components

The `PreferenceToggleMenuItem` components can be used with a `DropdownMenu` to implement a menu for changing preferences.

Also see the `MoreMenuDropdown` component from the `@wordpress/interface` package for implementing a more menu.

```

function MyEditorMenu() {
    return (
        <MoreMenuDropdown>
            { () => (
                <MenuGroup label={ __( 'Features' ) }>
                    <PreferenceToggleMenuItem
                        scope="namespace/editor-or-plugin-name"
                        name="myPreferenceName"
                        label={ __( 'My feature' ) }
                        info={ __( 'A really awesome feature' ) }
                        messageActivated={ __( 'My feature activated' ) }
                        messageDeactivated={ __( 'My feature deactivated' ) }
                    />
                </MenuGroup>
            ) }
        </MoreMenuDropdown>
    );
}

```

[API Reference](#)

[Actions](#)

The following set of dispatching action creators are available on the object returned by `wp.data.dispatch('core/preferences')`:

set

Returns an action object used in signalling that a preference should be set to a value

Parameters

- `scope` `string`: The preference scope (e.g. core/edit-post).
- `name` `string`: The preference name.
- `value` `*`: The value to set.

Returns

- `Object`: Action object.

setDefaults

Returns an action object used in signalling that preference defaults should be set.

Parameters

- `scope` `string`: The preference scope (e.g. core/edit-post).
- `defaults` `Object<string, *`: A key/value map of preference names to values.

Returns

- `Object`: Action object.

setPersistenceLayer

Sets the persistence layer.

When a persistence layer is set, the preferences store will:

- call `get` immediately and update the store state to the value returned.
- call `set` with all preferences whenever a preference changes value.

`setPersistenceLayer` should ideally be dispatched at the start of an application's lifecycle, before any other actions have been dispatched to the preferences store.

Parameters

- `persistenceLayer` `WPPreferencesPersistenceLayer`: The persistence layer.

Returns

- `Object`: Action object.

toggle

Returns an action object used in signalling that a preference should be toggled.

Parameters

- `scope` `string`: The preference scope (e.g. core/edit-post).
- `name` `string`: The preference name.

Selectors

The following selectors are available on the object returned by `wp.data.select('core/preferences')`:

get

Returns a boolean indicating whether a prefer is active for a particular scope.

Parameters

- `state` `Object`: The store state.
- `scope` `string`: The scope of the feature (e.g. core/edit-post).
- `name` `string`: The name of the feature.

Returns

- `*`: Is the feature enabled?

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](#).

First published

March 11, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/preferences”](#)

[Previous @wordpress/preferences-persistence](#) [Previous: @wordpress/preferences-persistence](#)
[Next @wordpress/prettier-config](#) [Next: @wordpress/prettier-config](#)

@wordpress/prettier-config

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

WordPress Prettier shareable config for [Prettier](#).

[Installation](#)

Install the module

```
$ npm install @wordpress/prettier-config --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

[Usage](#)

Add this to your package.json file:

```
"prettier": "@wordpress/prettier-config"
```

Alternatively, add this to .prettierrc file:

```
"@wordpress/prettier-config"
```

Or add this to `.prettierrc.js` file:

```
module.exports = require( '@wordpress/prettier-config' );
```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/prettier-config”](#)

[Previous @wordpress/preferences](#) [Previous: @wordpress/preferences](#)

[Next @wordpress/primitives](#) [Next: @wordpress/primitives](#)

@wordpress/primitives

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Primitives to be used cross-platform.

[Installation](#)

Install the module

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

Usage

```
import { SVG, Path } from '@wordpress/primitives';

const myElement = (
  <SVG
    width="18"
    height="18"
    viewBox="0 0 18 18"
    xmlns="http://www.w3.org/2000/svg"
  >
    <Path d="M4.5 915.6-5.7 1.4 1.5L7.3 914.2 4.2-1.4 1.5L4.5 9z" />
  </SVG>
);

;
```

*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.*

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/primitives”](#)

[Previous @wordpress/prettier-config](#) [Previous: @wordpress/prettier-config](#)
[Next @wordpress/priority-queue](#) [Next: @wordpress/priority-queue](#)

@wordpress/priority-queue

In this article

Table of Contents

- [Installation](#)

- [API](#)
 - [createQueue](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This module allows you to run a queue of callback while on the browser's idle time making sure the higher-priority work is performed first.

Installation

Install the module

```
npm install @wordpress/priority-queue --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

createQueue

Creates a context-aware queue that only executes the last task of a given context.

Usage

```
import { createQueue } from '@wordpress/priority-queue';

const queue = createQueue();

// Context objects.
const ctx1 = {};
const ctx2 = {};

// For a given context in the queue, only the last callback is executed.
queue.add( ctx1, () => console.log( 'This will be printed first' ) );
queue.add( ctx2, () => console.log( "This won't be printed" ) );
queue.add( ctx2, () => console.log( 'This will be printed second' ) );
```

Returns

- `WPPriorityQueue`: Queue object with `add`, `flush` and `reset` methods.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/priority-queue”](#)

[Previous @wordpress/primitives](#) [Previous: @wordpress/primitives](#)
[Next @wordpress/private-apis](#) [Next: @wordpress/private-apis](#)

@wordpress/private-apis

In this article

[Table of Contents](#)

- [Getting started](#)
- [Shipping experimental APIs](#)
- [Technical limitations](#)
- [Contributing to this package](#)
 - [Updating the consent string](#)

[↑ Back to top](#)

@wordpress/private-apis enables sharing __experimental APIs across @wordpress packages without [publicly exposing them to WordPress extenders](#).

Getting started

Every @wordpress package wanting to privately access or expose experimental APIs must opt-in to @wordpress/private-apis:

```
// In packages/block-editor/private-apis.js:  
import { __dangerousOptInToUnstableAPIsOnlyForCoreModules } from '@wordpress/core-data'  
export const { lock, unlock } =  
  __dangerousOptInToUnstableAPIsOnlyForCoreModules(  
    'I know using unstable features means my theme or plugin will inevitably break other packages.',  
    '@wordpress/block-editor' // Name of the package calling __dangerousOptInToUnstableAPIsOnlyForCoreModules  
    // (not the name of the package whose APIs you want to access)  
  );
```

Each package may only opt in once. The function name communicates that plugins are not supposed to use it.

The function will throw an error if the following conditions are not met:

1. The first argument must exactly match the required consent string: 'I know using unstable features means my theme or plugin will inevitably break in the next version of WordPress.'
2. The second argument must be a known @wordpress package that hasn't yet opted into @wordpress/private-apis

Once the opt-in is complete, the obtained lock() and unlock() utilities enable hiding __experimental APIs from the naked eye:

```
// Say this object is exported from a package:  
export const publicObject = {};  
  
// However, this string is internal and should not be publicly available:  
const __experimentalString = '__experimental information';  
  
// Solution: lock the string "inside" of the object:  
lock( publicObject, __experimentalString );  
  
// 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 ) );  
// "__experimental information"  
  
// lock() accepts all data types, not just strings:  
export const anotherObject = {};  
lock( anotherObject, function __experimentalFn() {} );  
console.log( unlock( anotherObject ) );  
// function __experimentalFn() {}
```

Use lock() and unlock() to privately distribute the __experimental APIs across @wordpress packages:

```
// In packages/package1/index.js:  
import { lock } from './lock-unlock';  
  
export const privateApis = {};  
/* Attach private data to the exported object */  
lock( privateApis, {  
    __experimentalFunction: function () {},  
} );  
  
// In packages/package2/index.js:  
import { privateApis } from '@wordpress/package1';  
import { unlock } from './lock-unlock';  
  
const { __experimentalFunction } = unlock( privateApis );
```

Shipping experimental APIs

See the [Experimental and Unstable APIs chapter of Coding Guidelines](#) to learn how `lock()` and `unlock()` can help you ship private experimental functions, arguments, components, properties, actions, selectors.

Technical limitations

A determined developer who would want to use the private experimental APIs at all costs would have to:

- Realize a private importing system exists
- Read the code where the risks would be spelled out in capital letters
- Explicitly type out he or she is aware of the consequences
- Pretend to register a `@wordpress` package (and trigger an error as soon as the real package is loaded)

Dangerously opting in to using these APIs by theme and plugin developers is not recommended. Furthermore, the WordPress Core philosophy to strive to maintain backward compatibility for third-party developers **does not apply** to experimental APIs registered via this package.

The consent string for opting in to these APIs may change at any time and without notice. This change will break existing third-party code. Such a change may occur in either a major or minor release.

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](#).

CODE IS POETRY

Updating the consent string

The consent string for unlocking private APIs is intended to change on a regular basis. To update the consent string:

1. Come up with a new consent string, the string should mention that themes or plugins opting in to unstable and private features will break in future versions of WordPress.
2. Ensure the consent string has not been used previously.
3. Append the new string to the history list below.
4. Replace the consent string in the following locations:
 - twice in the documentation above
 - in the `src/implementation.js` file of this package
 - in the `src/lock-unlock.js` file located in packages consuming private APIs
 - search the full code base for any other occurrences

Note: The consent string is not used for user facing content and as such should *not* be made translatable via the internationalization features of WordPress.

Updating the consent string is considered a task and can be done during the late stages of a WordPress release.

Consent string history

The final string in this list is the current version.

1. I know using unstable features means my plugin or theme will inevitably break on the next WordPress release.
2. I know using unstable features means my theme or plugin will inevitably break in the next version of WordPress.

First published

February 9, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/private-apis”](#)

[Previous @wordpress/priority-queue](#) [Previous: @wordpress/priority-queue](#)

[Next @wordpress/project-management-automation](#) [Next: @wordpress/project-management-automation](#)

@wordpress/project-management-automation

In this article

Table of Contents

- [Installation and usage](#)
- [API](#)
 - [Inputs](#)
 - [Outputs](#)
 - [Contributing to this package](#)

[↑ Back to top](#)

This is a [GitHub Action](#) which contains various automation to assist with managing the Gutenberg GitHub repository:

- [First Time Contributor](#): Adds the “First Time Contributor” label to pull requests merged on behalf of contributors that have not previously made a contribution, and prompts the user to

- link their GitHub account to their WordPress.org profile if necessary for release notes credit.
- [Add Milestone](#): Assigns the plugin release milestone to a pull request once it is merged.
- [Assign Fixed Issues](#): Adds assignee for issues which are marked to be “Fixed” by a pull request, and adds the “In Progress” label.

Installation and usage

To use the action, include it in your workflow configuration file:

```
on: pull_request
jobs:
  pull-request-automation:
    runs-on: ubuntu-latest
    steps:
      - uses: WordPress/gutenberg/packages/project-management-automation@v1
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
```

API

Inputs

- `github_token`: Required. GitHub API token to use for making API requests. This should be stored as a secret in the GitHub repository.

Outputs

None.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/project-management-automation”](#)

[Previous @wordpress/private-apis](#) Previous: [@wordpress/private-apis](#)

Next [@wordpress/react-i18n](#) Next: [@wordpress/react-i18n](#)

@wordpress/react-i18n

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [I18nProvider](#)
 - [useI18n](#)
 - [withI18n](#)
- [Contributing to this package](#)

[↑ Back to top](#)

React bindings for [@wordpress/i18n](#).

[Installation](#)

Install the module:

```
npm install @wordpress/react-i18n
```

*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](#)

[I18nProvider](#)

The `I18nProvider` should be mounted above any localized components:

Usage

```
import { createI18n } from '@wordpress/i18n';
import { I18nProvider } from '@wordpress/react-i18n';
const i18n = createI18n();

ReactDOM.render(
  <I18nProvider i18n={ i18n }>
    <App />
  </I18nProvider>,
```

```
    el
};
```

You can also instantiate the provider without the `i18n` prop. In that case it will use the default `I18n` instance exported from [@wordpress/i18n](#).

Parameters

- `props I18nProviderProps`: i18n provider props.

Returns

- `JSX.Element`: Children wrapped in the `I18nProvider`.

[useI18n](#)

React hook providing access to i18n functions. It exposes the `__`, `_x`, `_n`, `_nx`, `isRTL` and `hasTranslation` functions from [@wordpress/i18n](#). Refer to their documentation there.

Usage

```
import { useI18n } from '@wordpress/react-i18n';

function MyComponent() {
    const { __ } = useI18n();
    return __( 'Hello, world!' );
}
```

[withI18n](#)

React higher-order component that passes the i18n translate functions (the same set as exposed by the `useI18n` hook) to the wrapped component as props.

Usage

```
import { withI18n } from '@wordpress/react-i18n';

function MyComponent( { __ } ) {
    return __( 'Hello, world!' );
}

export default withI18n( MyComponent );
```

Parameters

- `InnerComponent ComponentType< P >`: React component to be wrapped and receive the i18n functions like `__`

Returns

- `FunctionComponent< PropsAndI18n< P > >`: The wrapped 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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/react-i18n”](#)

[Previous @wordpress/project-management-automation](#) [Previous: @wordpress/project-management-automation](#)

[Next @wordpress/readable-js-assets-webpack-plugin](#) [Next: @wordpress/readable-js-assets-webpack-plugin](#)

@wordpress/readable-js-assets-webpack-plugin

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
 - [Webpack](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Generate a readable non-minified JS file for each `.min.js` asset.

The end result is that for each JS entrypoint, we get a set of readable and non-minimized `.js` file and a minimized `.min.js`. This allows Gutenberg to follow the WordPress convention of adding a `.min.js` suffix to minimized JS files, while still providing a readable and unminimized files that play well with the WordPress i18n machinery.

Consult the [webpack website](#) for additional information on webpack concepts.

Installation

Install the module

```
npm install @wordpress/readable-js-assets-webpack-plugin --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It also requires webpack 4.8.3 and newer. It is not compatible with older versions.

Usage

Webpack

Use this plugin as you would other webpack plugins:

```
// webpack.config.js
const ReadableJsAssetsWebpackPlugin = require( '@wordpress/readable-js-ass

module.exports = {
    // ...snip
    plugins: [ new ReadableJsAssetsWebpackPlugin() ] ,
};
```

Note:

- Multiple instances of the plugin are not supported and may produce unexpected results;
- It assumes your webpack pipeline is already generating a `.min.js` JS asset file for each JS entry-point.

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](#).

First published

June 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/readable-js-assets-webpack-plugin](#)

[Previous: @wordpress/react-i18n](#) [Previous: @wordpress/react-i18n](#)

[Next: @wordpress/redux-routine](#) [Next: @wordpress/redux-routine](#)

@wordpress/redux-routine

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [API](#)
 - [default](#)
- [Motivation](#)
- [Testing](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Redux middleware for generator coroutines.

[Installation](#)

Install Node if you do not already have it available.

Install the module to your project using npm:

```
npm install @wordpress/redux-routine
```

`@wordpress/redux-routine` leverages both Promises and Generators, two modern features of the JavaScript language. If you need to support older browsers (Internet Explorer 11 or earlier), you will need to provide your own polyfills.

[Usage](#)

The default export of `@wordpress/redux-routine` is a function which, given an object of control handlers, returns a Redux middleware function.

For example, consider a common case where we need to issue a network request. We can define the network request as a control handler when creating our middleware.

```
import { combineReducers, createStore, applyMiddleware } from 'redux';
import createMiddleware from '@wordpress/redux-routine';

const middleware = createMiddleware( {
  async FETCH_JSON( action ) {
    const response = await window.fetch( action.url );
    return response.json();
  },
} );

function temperature( state = null, action ) {
  switch ( action.type ) {
```

```

        case 'SET_TEMPERATURE':
            return action.temperature;
    }

    return state;
}

const reducer = combineReducers( { temperature } );
const store = createStore( reducer, applyMiddleware( middleware ) );

function* retrieveTemperature() {
    const result = yield { type: 'FETCH_JSON', url: 'https://' };
    return { type: 'SET_TEMPERATURE', temperature: result.value };
}

store.dispatch( retrieveTemperature() );

```

In this example, when we dispatch `retrieveTemperature`, it will trigger the control handler to take effect, issuing the network request and assigning the result into the `result` variable. Only once the request has completed does the action creator proceed to return the `SET_TEMPERATURE` action type.

[API](#)

[default](#)

Creates a Redux middleware, given an object of controls where each key is an action type for which to act upon, the value a function which returns either a promise which is to resolve when evaluation of the action should continue, or a value. The value or resolved promise value is assigned on the return value of the yield assignment. If the control handler returns undefined, the execution is not continued.

Parameters

- `controls` Record<string, (value: import('redux').AnyAction) => Promise<boolean> | boolean>: Object of control handlers.

Returns

- `import('redux').Middleware`: Co-routine runtime

[Motivation](#)

`@wordpress/redux-routine` shares many of the same motivations as other similar generator-based Redux side effects solutions, including `redux-saga`. Where it differs is in being less opinionated by virtue of its minimalism. It includes no default controls, offers no tooling around splitting logic flows, and does not include any error handling out of the box. This is intended in promoting approachability to developers who seek to bring asynchronous or conditional continuation flows to their applications without a steep learning curve.

The primary motivations include, among others:

- **Testability:** Since an action creator yields plain action objects, the behavior of their resolution can be easily substituted in tests.
- **Controlled flexibility:** Control flows can be implemented without sacrificing the expressiveness and intentionality of an action type. Other solutions like thunks or promises promote ultimate flexibility, but at the expense of maintainability manifested through deep coupling between action types and incidental implementation.
- A **common domain language** for expressing data flows: Since controls are centrally defined, it requires the conscious decision on the part of a development team to decide when and how new control handlers are added.

Testing

Since your action creators will return an iterable generator of plain action objects, they are trivial to test.

Consider again our above example:

```
function* retrieveTemperature() {
  const result = yield { type: 'FETCH_JSON', url: 'https://' };
  return { type: 'SET_TEMPERATURE', temperature: result.value };
}
```

A test case (using Node's `assert` built-in module) may be written as:

```
import { deepEqual } from 'assert';

const action = retrieveTemperature();

deepEqual( action.next().value, {
  type: 'FETCH_JSON',
  url: 'https://',
} );

const jsonResult = { value: 10 };
deepEqual( action.next( jsonResult ).value, {
  type: 'SET_TEMPERATURE',
  temperature: 10,
} );
```

If your action creator does not assign the yielded result into a variable, you can also use `Array.from` to create an array from the result of the action creator.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/redux-routine”](#)

[Previous @wordpress/readable-js-assets-webpack-plugin](#) Previous: [@wordpress/readable-js-assets-webpack-plugin](#)

Next [@wordpress/reusable-blocks](#) Next: [@wordpress/reusable-blocks](#)

@wordpress/reusable-blocks

In this article

[Table of Contents](#)

- [Installation](#)
- [How it works](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Reusable blocks components and logic.

[Installation](#)

Install the module

```
npm install @wordpress/reusable-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.*

[How it works](#)

This experimental module provides support for reusable blocks.

Reusable blocks are WordPress entities and the following is enough to ensure they are available in the inserter:

```
const { __experimentalReusableBlocks } = useSelect( ( select ) =>
  select( 'core' ).getEntityRecords( 'postType', 'wp_block' )
);
```

```

return (
  <BlockEditorProvider
    value={ blocks }
    onInput={ onInput }
    onChange={ onChange }
    settings={ {
      ...settings,
      __experimentalReusableBlocks,
    } }
    { ...props }
  />
);

```

With the above configuration management features (such as creating new reusable blocks) are still missing from the editor. Enter [@wordpress/reusable-blocks](#):

```

import { ReusableBlocksMenuItems } from '@wordpress/reusable-blocks';

const { __experimentalReusableBlocks } = useSelect( ( select ) =>
  select( 'core' ).getEntityRecords( 'postType', 'wp_block' )
);

return (
  <BlockEditorProvider
    value={ blocks }
    onInput={ onInput }
    onChange={ onChange }
    settings={ {
      ...settings,
      __experimentalReusableBlocks,
    } }
    { ...props }
  >
    <ReusableBlocksMenuItems />
    { children }
  </BlockEditorProvider>
);

```

This package also provides convenient utilities for managing reusable blocks through redux actions:

```

import { store as reusableBlocksStore } from '@wordpress/reusable-blocks';

function MyConvertToStaticButton( { clientId } ) {
  const { __experimentalConvertBlockToStatic } = useDispatch(
    reusableBlocksStore
  );
  return (
    <button
      onClick={ () => __experimentalConvertBlockToStatic( clientId ) }
    >
      Convert to static
    </button>
  );
}

```

```

    );
}

function MyConvertToReusableButton( { clientId } ) {
  const { __experimentalConvertBlocksToReusable } = useDispatch(
    reusableBlocksStore
  );
  return (
    <button
      onClick={ () =>
        __experimentalConvertBlocksToReusable( [ clientId ] )
      }
    >
      Convert to reusable
    </button>
  );
}

function MyDeleteReusableBlockButton( { id } ) {
  const { __experimentalDeleteReusableBlock } = useDispatch(
    reusableBlocksStore
  );
  return (
    <button onClick={ () => __experimentalDeleteReusableBlock( id ) }>
      Delete reusable block
    </button>
  );
}

```

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/reusable-blocks](#)"

[Previous @wordpress/redux-routine](#) [Previous: @wordpress/redux-routine](#)
[Next @wordpress/rich-text](#) [Next: @wordpress/rich-text](#)

@wordpress/rich-text

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
 - [The RichTextValue object](#)
 - [Selections](#)
- [API](#)
 - [applyFormat](#)
 - [concat](#)
 - [create](#)
 - [getActiveFormat](#)
 - [getActiveFormats](#)
 - [getActiveObject](#)
 - [getTextContent](#)
 - [insert](#)
 - [insertObject](#)
 - [isCollapsed](#)
 - [isEmpty](#)
 - [join](#)
 - [registerFormatType](#)
 - [remove](#)
 - [removeFormat](#)
 - [replace](#)
 - [RichTextData](#)
 - [RichTextValue](#)
 - [slice](#)
 - [split](#)
 - [store](#)
 - [toggleFormat](#)
 - [toHTMLString](#)
 - [unregisterFormatType](#)
 - [useAnchor](#)
 - [useAnchorRef](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This module contains helper functions to convert HTML or a DOM tree into a rich text value and back, and to modify the value with functions that are similar to `String` methods, plus some additional ones for formatting.

Installation

Install the module

```
npm install @wordpress/rich-text
```

*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.*

Usage

The Rich Text package is designed to aid in the manipulation of plain text strings in order that they can represent complex formatting.

By using a `RichTextValue` value object (referred to from here on as `value`) it is possible to separate text from formatting, thereby affording the ability to easily search and manipulate rich formats.

Examples of rich formats include:

- bold, italic, superscript (etc)
- links
- unordered/ordered lists

The `RichTextValue` object

The `value` object is comprised of the following:

- `text` – the string of text to which rich formats are to be applied.
- `formats` – a sparse array of the same length as `text` that is filled with [`formats`](#) (e.g. `core/link`, `core/bold` etc.) at the positions where the text is formatted.
- `start` – an index in the `text` representing the *start* of the currently active selection.
- `end` – an index in the `text` representing the *end* of the currently active selection.

You should not attempt to create your own `value` objects. Rather you should rely on the built in methods of the `@wordpress/rich-text` package to build these for you.

It is important to understand how a `value` represents richly formatted text. Here is an example to illustrate.

If `text` is formatted from position 2-5 in bold (`core/bold`) and from position 2-8 with a link (`core/link`), then you'll find:

- arrays within the sparse array at positions 2-5 that include the `core/bold` format
- arrays within the sparse array at positions 2-8 that include the `core/link` format

Here's how that would look:

```
{  
  text: 'Hello world', // length 11  
  formats: [  
    [], // 0  
    [],  
    [ // 2  
      {  
        type: 'core/bold',  
      },  
      {  
        type: 'core/link',  
      },  
      {  
        type: 'core/link',  
      }  
    ]  
  ]  
}
```

```

        }
    ],
    [
      {
        type: 'core/bold',
      },
      {
        type: 'core/link',
      }
    ],
    [
      {
        type: 'core/bold',
      },
      {
        type: 'core/link',
      }
    ],
    [
      {
        type: 'core/bold',
      },
      {
        type: 'core/link',
      }
    ],
    [
      []
    ], // 9
    [
      []
    ], // 10
    [
      []
    ], // 11
  ]
}

```

Selections

Let's continue to consider the above example with the text `Hello world`.

If, as a user, I make a selection of the word `Hello` this would result in a value object with `start` and `end` as `0` and `5` respectively.

In general, this is useful for knowing which portion of the text is selected. However, we need to consider that selections may also be “collapsed”.

Collapsed selections

A collapsed selection is one where `start` and `end` values are *identical* (e.g. `start: 4, end: 4`). This happens when no characters are selected, but there is a caret present. This most often occurs when a user places the cursor/caret within a string of text but does not make a selection.

Given that the selection has no “range” (i.e. there is no difference between `start` and `end` indices), finding the currently selected portion of text from collapsed values can be challenging.

API

applyFormat

Apply a format object to a Rich Text value from the given `startIndex` to the given `endIndex`. Indices are retrieved from the selection if none are provided.

Parameters

- `value RichTextValue`: Value to modify.
- `format RichTextFormat`: Format to apply.
- `startIndex [number]`: Start index.
- `endIndex [number]`: End index.

Returns

- `RichTextValue`: A new value with the format applied.

concat

Combine all Rich Text values into one. This is similar to `String.prototype.concat`.

Parameters

- `values ...RichTextValue`: Objects to combine.

Returns

- `RichTextValue`: A new value combining all given records.

create

Create a RichText value from an Element tree (DOM), an HTML string or a plain text string, with optionally a Range object to set the selection. If called without any input, an empty value will be created. The optional functions can be used to filter out content.

A value will have the following shape, which you are strongly encouraged not to modify without the use of helper functions:

```

{
  text: string,
  formats: Array,
  replacements: Array,
  ?start: number,
  ?end: number,
}

```

As you can see, text and formatting are separated. `text` holds the text, including any replacement characters for objects and lines. `formats`, `objects` and `lines` are all sparse arrays of the same length as `text`. It holds information about the formatting at the relevant text indices. Finally `start` and `end` state which text indices are selected. They are only provided if a Range was given.

Parameters

- `$1 [Object]`: Optional named arguments.
- `$1.element [Element]`: Element to create value from.
- `$1.text [string]`: Text to create value from.
- `$1.html [string]`: HTML to create value from.
- `$1.range [Range]`: Range to create value from.
- `$1.__unstableIsEditableTree [boolean]`:

Returns

- `RichTextValue`: A rich text value.

[getActiveFormat](#)

Gets the format object by type at the start of the selection. This can be used to get e.g. the URL of a link format at the current selection, but also to check if a format is active at the selection. Returns undefined if there is no format at the selection.

Parameters

- `value RichTextValue`: Value to inspect.
- `formatType string`: Format type to look for.

Returns

- `RichTextFormat | undefined`: Active format object of the specified type, or undefined.

[getActiveFormats](#)

Gets the all format objects at the start of the selection.

Parameters

- `value RichTextValue`: Value to inspect.
- `EMPTY_ACTIVE_FORMATS Array`: Array to return if there are no active formats.

Returns

- `RichTextFormatList`: Active format objects.

[getActiveObject](#)

Gets the active object, if there is any.

Parameters

- *value RichTextValue*: Value to inspect.

Returns

- *RichTextFormat | void*: Active object, or undefined.

[getTextContent](#)

Get the textual content of a Rich Text value. This is similar to `Element.textContent`.

Parameters

- *value RichTextValue*: Value to use.

Returns

- *string*: The text content.

[insert](#)

Insert a Rich Text value, an HTML string, or a plain text string, into a Rich Text value at the given `startIndex`. Any content between `startIndex` and `endIndex` will be removed. Indices are retrieved from the selection if none are provided.

Parameters

- *value RichTextValue*: Value to modify.
- *valueToInsert RichTextValue | string*: Value to insert.
- *startIndex [number]*: Start index.
- *endIndex [number]*: End index.

Returns

- *RichTextValue*: A new value with the value inserted.

[insertObject](#)

Insert a format as an object into a Rich Text value at the given `startIndex`. Any content between `startIndex` and `endIndex` will be removed. Indices are retrieved from the selection if none are provided.

Parameters

- *value RichTextValue*: Value to modify.
- *formatToInsert RichTextFormat*: Format to insert as object.
- *startIndex [number]*: Start index.
- *endIndex [number]*: End index.

Returns

- `RichTextValue`: A new value with the object inserted.

isCollapsed

Check if the selection of a Rich Text value is collapsed or not. Collapsed means that no characters are selected, but there is a caret present. If there is no selection, `undefined` will be returned. This is similar to `window.getSelection().isCollapsed()`.

Parameters

- `props RichTextValue`: The rich text value to check.
- `props.start RichTextValue['start']`:
- `props.end RichTextValue['end']`:

Returns

- `boolean | undefined`: True if the selection is collapsed, false if not, `undefined` if there is no selection.

isEmpty

Check if a Rich Text value is Empty, meaning it contains no text or any objects (such as images).

Parameters

- `value RichTextValue`: Value to use.

Returns

- `boolean`: True if the value is empty, false if not.

join

Combine an array of Rich Text values into one, optionally separated by `separator`, which can be a Rich Text value, HTML string, or plain text string. This is similar to `Array.prototype.join`.

Parameters

- `values Array<RichTextValue>`: An array of values to join.
- `separator [string|RichTextValue]`: Separator string or value.

Returns

- `RichTextValue`: A new combined value.

registerFormatType

Registers a new format provided a unique name and an object defining its behavior.

Parameters

- *name* `string`: Format name.
- *settings* `WPFormat`: Format settings.

Returns

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

[remove](#)

Remove content from a Rich Text value between the given `startIndex` and `endIndex`. Indices are retrieved from the selection if none are provided.

Parameters

- *value* `RichTextValue`: Value to modify.
- *startIndex* `[number]`: Start index.
- *endIndex* `[number]`: End index.

Returns

- `RichTextValue`: A new value with the content removed.

[removeFormat](#)

Remove any format object from a Rich Text value by type from the given `startIndex` to the given `endIndex`. Indices are retrieved from the selection if none are provided.

Parameters

- *value* `RichTextValue`: Value to modify.
- *formatType* `string`: Format type to remove.
- *startIndex* `[number]`: Start index.
- *endIndex* `[number]`: End index.

Returns

- `RichTextValue`: A new value with the format applied.

[replace](#)

Search a Rich Text value and replace the match(es) with `replacement`. This is similar to `String.prototype.replace`.

Parameters

- *value* `RichTextValue`: The value to modify.
- *pattern* `RegExp | string`: A `RegExp` object or literal. Can also be a string. It is treated as a verbatim string and is not interpreted as a regular expression. Only the first occurrence will be replaced.
- *replacement* `Function | string`: The match or matches are replaced with the specified or the value returned by the specified function.

Returns

- `RichTextValue`: A new value with replacements applied.

[RichTextData](#)

The `RichTextData` class is used to instantiate a wrapper around rich text values, with methods that can be used to transform or manipulate the data.

- Create an empty instance: `new RichTextData()`.
- Create one from an html string: `RichTextData.fromHTMLString('hello')`.
- Create one from a wrapper `HTMLElement`: `RichTextData.fromHTMLElement(document.querySelector('p'))`.
- Create one from plain text: `RichTextData.fromPlainText('1\n2')`.
- Create one from a rich text value: `new RichTextData({ text: '...', formats: [...] })`.

[RichTextValue](#)

An object which represents a formatted string. See main `@wordpress/rich-text` documentation for more information.

[slice](#)

Slice a Rich Text value from `startIndex` to `endIndex`. Indices are retrieved from the selection if none are provided. This is similar to `String.prototype.slice`.

Parameters

- `value RichTextValue`: Value to modify.
- `startIndex [number]`: Start index.
- `endIndex [number]`: End index.

Returns

- `RichTextValue`: A new extracted value.

[split](#)

Split a Rich Text value in two at the given `startIndex` and `endIndex`, or split at the given separator. This is similar to `String.prototype.split`. Indices are retrieved from the selection if none are provided.

Parameters

- `value RichTextValue`:
- `string [number|string]`: Start index, or string at which to split.

Returns

- `Array<RichTextValue>|undefined`: An array of new values.

[store](#)

Store definition for the rich-text namespace.

Related

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

Type

- Object

[toggleFormat](#)

Toggles a format object to a Rich Text value at the current selection.

Parameters

- *value* RichTextValue: Value to modify.
- *format* RichTextFormat: Format to apply or remove.

Returns

- RichTextValue: A new value with the format applied or removed.

[toHTMLString](#)

Create an HTML string from a Rich Text value.

Parameters

- \$1 Object: Named arguments.
- \$1.value RichTextValue: Rich text value.

Returns

- string: HTML string.

[unregisterFormatType](#)

Unregisters a format.

Parameters

- *name* string: Format name.

Returns

- WPFormat | undefined: The previous format value, if it has been successfully unregistered; otherwise undefined.

[useAnchor](#)

This hook, to be used in a format type's Edit component, returns the active element that is formatted, or a virtual element for the selection range if no format is active. The returned value is meant to be used for positioning UI, e.g. by passing it to the `Popover` component via the `anchor` prop.

Parameters

- `$1 Object`: Named parameters.
- `$1.editableContentElement HTMLElement | null`: The element containing the editable content.
- `$1.settings WPFormat=`: The format type's settings.

Returns

- `Element | VirtualAnchorElement | undefined | null`: The active element or selection range.

[useAnchorRef](#)

This hook, to be used in a format type's Edit component, returns the active element that is formatted, or the selection range if no format is active. The returned value is meant to be used for positioning UI, e.g. by passing it to the `Popover` component.

Parameters

- `$1 Object`: Named parameters.
- `$1.refRefObject<HTMLElement>`: React ref of the element containing the editable content.
- `$1.value RichTextValue`: Value to check for selection.
- `$1.settings WPFormat`: The format type's settings.

Returns

- `Element | Range`: The active element or selection range.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/rich-text”](#)

[Previous @wordpress/reusable-blocks](#) [Previous: @wordpress/reusable-blocks](#)

[Next @wordpress/router](#) [Next: @wordpress/router](#)

@wordpress/router

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [privateApis](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Router is a generic package that allows to use browser routing in WordPress packages.

[Installation](#)

Install the module

```
npm install @wordpress/router --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](#)

[privateApis](#)

Undocumented declaration.

[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](#).

First published

April 27, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/router](#)

[Previous @wordpress/rich-text](#) [Previous: @wordpress/rich-text](#)

[Next @wordpress/scripts](#) [Next: @wordpress/scripts](#)

@wordpress/scripts

In this article

[Table of Contents](#)

- [Installation](#)
- [Setup](#)
- [Automatic block.json detection and the source code directory](#)
- [Updating to New Release](#)
- [Available Scripts](#)
 - [build](#)
 - [check-engines](#)
 - [check-licenses](#)
 - [format](#)
 - [lint-js](#)
 - [lint-pkg-json](#)
 - [lint-md-docs](#)
 - [lint-style](#)
 - [packages-update](#)
 - [plugin-zip](#)
 - [start](#)
 - [test-e2e](#)
 - [test-unit-js](#)
 - [test-playwright](#)
- [Passing Node.js options](#)
 - [Debugging tests](#)
- [Advanced Usage](#)
 - [Working with build scripts](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This is a collection of reusable scripts tailored for WordPress development. For convenience, every tool provided in this package comes with an integrated recommended configuration.

When working seamlessly, sophisticated command-line interfaces help to turn work with a project into a more pleasant experience. However, it's a misleading assumption that developers can easily pick the proper tools in the first place and then ensure that they play along with each

other, including all their extensions. Besides, it's still not enough because developers are left on their own to keep all configurations and dependent tools up to date. This problem multiplies when they support more than one project which shares the same setup.

Fortunately, there is a pattern that can simplify maintainers life – reusable scripts. The idea boils down to moving all the necessary configurations and scripts to one single tool dependency. In most cases, it should be possible to accomplish all tasks using the default settings, but some customization is allowed, too. With all that in place, updating all projects should become a very straightforward task.

This package is inspired by [react-scripts](#) and [kcd-scripts](#).

Installation

You only need to install one npm module:

```
npm install @wordpress/scripts --save-dev
```

Note: This package requires Node.js 14.0.0 or later, and npm 6.14.4 or later. It is not compatible with older versions.

Setup

This package offers a command-line interface and exposes a binary called `wp-scripts` so you can call it directly with `npx` – an npm package runner. However, this module is designed to be configured using the `scripts` section in the `package.json` file of your project. This comprehensive example demonstrates the most of the capabilities included.

Example:

```
{
  "scripts": {
    "build": "wp-scripts build",
    "check-engines": "wp-scripts check-engines",
    "check-licenses": "wp-scripts check-licenses",
    "format": "wp-scripts format",
    "lint:css": "wp-scripts lint-style",
    "lint:js": "wp-scripts lint-js",
    "lint:md:docs": "wp-scripts lint-md-docs",
    "lint:pkg-json": "wp-scripts lint-pkg-json",
    "packages-update": "wp-scripts packages-update",
    "plugin-zip": "wp-scripts plugin-zip",
    "start": "wp-scripts start",
    "test:e2e": "wp-scripts test-e2e",
    "test:unit": "wp-scripts test-unit-js"
  }
}
```

It might also be a good idea to get familiar with the [JavaScript Build Setup tutorial](#) for setting up a development environment to use ESNext syntax. It gives a very in-depth explanation of how to use the `build` and `start` scripts.

Automatic block.json detection and the source code directory

When using the `start` or `build` commands, the source code directory (the default is `./src`) and its subdirectories are scanned for the existence of `block.json` files. If one or more are found, they are treated as entry points and will be output into corresponding folders in the `build` directory. This allows for the creation of multiple blocks that use a single build process. The source directory can be customized using the `--webpack-src-dir` flag and the output directory with the `--output-path` flag.

Updating to New Release

To update an existing project to a new version of `@wordpress/scripts`, open the [changelog](#), find the version you're currently on (check `package.json` in the top-level directory of your project), and apply the migration instructions for the newer versions.

In most cases bumping the `@wordpress/scripts` version in `package.json` and running `npm install` in the root folder of your project should be enough, but it's good to check the [changelog](#) for potential breaking changes. There is also `packages-update` script included in this package that aims to automate the process of updating WordPress dependencies in your projects.

We commit to keeping the breaking changes minimal so you can upgrade `@wordpress/scripts` as seamless as possible.

Available Scripts

build

Transforms your code according the configuration provided so it's ready for production and optimized for the best performance.

This script exits after producing a single build. For incremental builds, better suited for development, see the [start](#) script.

The entry points for your project get detected by scanning all script fields in `block.json` files located in the `src` directory. The script fields in `block.json` should pass relative paths to `block.json` in the same folder.

Example:

```
{  
  "editorScript": "file:index.js",  
  "script": "file:script.js",  
  "viewScript": "file:view.js"  
}
```

The fallback entry point is `src/index.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) in case there is no `block.json` file found. In that scenario, the output generated will be written to `build/index.js`.

Example:

```
{
  "scripts": {
    "build": "wp-scripts build",
    "build:custom": "wp-scripts build entry-one.js entry-two.js --outp
    "build:copy-php": "wp-scripts build --webpack-copy-php",
    "build:custom-directory": "wp-scripts build --webpack-src-dir=cust
  }
}
```

This is how you execute the script with presented setup:

- `npm run build` – builds the code for production.
- `npm run build:custom` – builds the code for production with two entry points and a custom output directory. Paths for custom entry points are relative to the project root.
- `npm run build:copy-php` – builds the code for production and opts into copying all PHP files from the `src` directory and its subfolders to the output directory. By default, only PHP files listed in the `render` field in the detected `block.json` files get copied.
- `npm run build:custom-directory` – builds the code for production using the `custom-directory` as the source code directory.

This script automatically use the optimized config but sometimes you may want to specify some custom options:

- `--webpack-bundle-analyzer` – enables visualization for the size of webpack output files with an interactive zoomable treemap.
- `--webpack-copy-php` – enables copying all PHP files from the source directory (default is `src`) and its subfolders to the output directory.
- `--webpack-no-externals` – disables scripts' assets generation, and omits the list of default externals.
- `--webpack-src-dir` – Allows customization of the source code directory. Default is `src`.
- `--output-path` – Allows customization of the output directory. Default is `build`.

Experimental support for the `block.json` `viewModule` field is available via the `--experimental-modules` option. With this option enabled, script and module fields will all be compiled. The `viewModule` field is analogous to the `viewScript` field, but will compile a module and should be registered in WordPress using the Modules API.

Advanced information

This script uses [webpack](#) behind the scenes. It'll look for a webpack config in the top-level directory of your package and will use it if it finds one. If none is found, it'll use the default config provided by `@wordpress/scripts` packages. Learn more in the [Advanced Usage](#) section.

[check-engines](#)

Checks if the current node, `npm` (or `yarn`) versions match the given [semantic version](#) ranges. If the given version is not satisfied, information about installing the needed version is printed and the program exits with an error code.

Example:

```
{
  "scripts": {
    "check-engines": "wp-scripts check-engines"
  }
}
```

This is how you execute the script with presented setup:

- `npm run check-engines` – checks installed version of node and npm.

Advanced information

It uses [check-node-version](#) behind the scenes with the recommended configuration provided. The default requirements are set to the same Node.js and npm versions as listed in the [installation](#) section for this package. You can specify your own ranges as described in [check-node-version docs](#). Learn more in the [Advanced Usage](#) section.

[check-licenses](#)

Validates that all dependencies of a project are compatible with the project's own license.

Example:

```
{
  "scripts": {
    "check-licenses": "wp-scripts check-licenses --prod --gpl2 --ignore"
  }
}
```

Flags:

- `--prod` (or `--production`): When present, validates only `dependencies` and not `devDependencies`
- `--dev` (or `--development`): When present, validates only `devDependencies` and not `dependencies`
- `--gpl2`: Validates against [GPLv2 license compatibility](#)
- `--ignore=a,b,c`: A comma-separated set of package names to ignore for validation. This is intended to be used primarily in cases where a dependency's `license` field is malformed. It's assumed that any `ignored` package argument would be manually vetted for compatibility by the project owner.

[format](#)

It helps to enforce coding style guidelines for your files (enabled by default for JavaScript, JSON, TypeScript, YAML) by formatting source code in a consistent way.

Example:

```
{
  "scripts": {
    "format": "wp-scripts format",
    "format:src": "wp-scripts format ./src"
  }
}
```

This is how you execute the script with presented setup:

- `npm run format` – formats files in the entire project's directories.
- `npm run format:src` – formats files in the project's `src` subfolder's directories.

When you run commands similar to the `npm run format:src` example above, you can provide a file, a directory, or `glob` syntax or any combination of them.

By default, files located in `build`, `node_modules`, and `vendor` folders are ignored. You can customize the list of ignored files and directories by adding them to a `.prettierrcignore` file in your project.

[lint-js](#)

Helps enforce coding style guidelines for your JavaScript and TypeScript files.

Example:

```
{  
  "scripts": {  
    "lint:js": "wp-scripts lint-js",  
    "lint:js:src": "wp-scripts lint-js ./src"  
  }  
}
```

This is how you execute the script with presented setup:

- `npm run lint:js` – lints JavaScript and TypeScript files in the entire project's directories.
- `npm run lint:js:src` – lints JavaScript and TypeScript files in the project's `src` subfolder's directories.

When you run commands similar to the `npm run lint:js:src` example above, you can provide a file, a directory, or `glob` syntax or any combination of them. See [more examples](#).

By default, files located in `build`, `node_modules`, and `vendor` folders are ignored.

Advanced information

It uses [eslint](#) with the set of recommended rules defined in [@wordpress/eslint-plugin](#) npm package. You can override default rules with your own as described in [eslint docs](#). Learn more in the [Advanced Usage](#) section.

[lint-pkg-json](#)

Helps enforce standards for your `package.json` files.

Example:

```
{  
  "scripts": {  
    "lint:pkg-json": "wp-scripts lint-pkg-json",  
    "lint:pkg-json:src": "wp-scripts lint-pkg-json ./src"  
  }  
}
```

```
}
```

This is how you execute those scripts using the presented setup:

- `npm run lint:pkg-json` – lints `package.json` file in the entire project's directories.
- `npm run lint:pkg-json:src` – lints `package.json` file in the project's `src` subfolder's directories.

When you run commands similar to the `npm run lint:pkg-json:src` example above, you can provide one or multiple directories to scan as well. See [more examples](#).

By default, files located in `build`, `node_modules`, and `vendor` folders are ignored.

Advanced information

It uses [npm-package-json-lint](#) with the set of recommended rules defined in [@wordpress/npm-package-json-lint-config](#) npm package. You can override default rules with your own as described in [npm-package-json-lint wiki](#). Learn more in the [Advanced Usage](#) section.

[lint-md-docs](#)

Uses `markdownlint` to lint the markup of markdown files to enforce standards.

Example:

```
{
  "scripts": {
    "lint:md:docs": "wp-scripts lint-md-docs"
  }
}
```

This is how you execute the script with presented setup:

- `npm run lint:md:docs` – lints markdown files in the entire project's directories.

By default, files located in `build`, `node_modules`, and `vendor` folders are ignored.

Advanced information

It uses [markdownlint](#) with the [.markdownlint.json](#) configuration. This configuration tunes the linting rules to match WordPress standard, you can override with your own config, see [markdownlint-cli](#) for command-line parameters.

[lint-style](#)

Helps enforce coding style guidelines for your style files.

Example:

```
{
  "scripts": {
    "lint:style": "wp-scripts lint-style",
  }
}
```

```
        "lint:css:src": "wp-scripts lint-style 'src/**/*.css'"
    }
}
```

This is how you execute the script with presented setup:

- `npm run lint:style` – lints CSS, PCSS, and SCSS files in the entire project's directories.
- `npm run lint:css:src` – lints only CSS files in the project's `src` subfolder's directories.

When you run commands similar to the `npm run lint:css:src` example above, be sure to include the quotation marks around file globs. This ensures that you can use the powers of [globby](#) (like the `**` globstar) regardless of your shell. See [more examples](#).

By default, files located in `build`, `node_modules`, and `vendor` folders are ignored.

Advanced information

It uses [stylelint](#) with the [@wordpress/stylelint-config](#) configuration per the [WordPress CSS Coding Standards](#). You can override them with your own rules as described in [stylelint user guide](#). Learn more in the [Advanced Usage](#) section.

[packages-update](#)

Updates the WordPress packages used in the project to their latest version.

Example:

```
{
  "scripts": {
    "packages-update": "wp-scripts packages-update",
    "postpackages-update": "npm run build"
  }
}
```

This script provides the following custom options:

- `--dist-tag` – allows specifying a custom dist-tag when updating npm packages. Defaults to `latest`. This is especially useful when using [@wordpress/dependency-extraction-webpack-plugin](#). It lets installing the npm dependencies at versions used by the given WordPress major version for local testing, etc. Example: `wp-scripts packages-update --dist-tag=wp-6.0`.

Advanced information

The command detects project dependencies that have name starting with `@wordpress/` by scanning the `package.json` file. By default, it executes `npm install @wordpress/package1@latest @wordpress/package2@latest ... --save` to change the package versions to the latest one. You can chose a different dist-tag than `latest` by using the `--dist-tag` option when running the command.

plugin-zip

Creates a zip file for a WordPress plugin.

Example:

```
{  
  "scripts": {  
    "plugin-zip": "wp-scripts plugin-zip"  
  }  
}
```

By default, it uses [Plugin Handbook best practices](#) to discover files.

Advanced information

In the case where the plugin author wants to customize the files included in the zip file, they can provide the `files` field in the `package.json` file as documented in the [npm-packlist](#) package, example:

```
{  
  "files": [ "dir" ]  
}
```

It reuses the same logic as `npm pack` command to create an npm package tarball.

start

Transforms your code according the configuration provided so it's ready for development. The script will automatically rebuild if you make changes to the code, and you will see the build errors in the console.

For single builds, better suited for production, see the [build](#) script.

The entry points for your project get detected by scanning all script fields in `block.json` files located in the `src` directory. The script fields in `block.json` should pass relative paths to `block.json` in the same folder.

Example:

```
{  
  "editorScript": "file:index.js",  
  "script": "file:script.js",  
  "viewScript": "file:view.js"  
}
```

The fallback entry point is `src/index.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) in case there is no `block.json` file found. In that scenario, the output generated will be written to `build/index.js`.

Example:

```
{  
  "scripts": {
```

```

    "start": "wp-scripts start",
    "start:hot": "wp-scripts start --hot",
    "start:custom": "wp-scripts start entry-one.js entry-two.js --outp
    "start:copy-php": "wp-scripts start --webpack-copy-php",
    "start:custom-directory": "wp-scripts start --webpack-src-dir=cust
  }
}

```

This is how you execute the script with presented setup:

- `npm start` – starts the build for development.
- `npm run start:hot` – starts the build for development with “Fast Refresh”. The page will automatically reload if you make changes to the files.
- `npm run start:custom` – starts the build for development which contains two entry points and a custom output directory. Paths for custom entry points are relative to the project root.
- `npm run start:copy-php` – starts the build for development and opts into copying all PHP files from the `src` directory and its subfolders to the output directory. By default, only PHP files listed in the `render` field in the detected `block.json` files get copied.
- `npm run start:custom-directory` – builds the code for production using the `custom-directory` as the source code directory.

This script automatically use the optimized config but sometimes you may want to specify some custom options:

- `--hot` – enables “Fast Refresh”. The page will automatically reload if you make changes to the code. *For now, it requires that WordPress has the [SCRIPT_DEBUG](#) flag enabled and the [Gutenberg](#) plugin installed.*
- `--no-watch` – Starts the build for development without starting the watcher.
- `--webpack-bundle-analyzer` – enables visualization for the size of webpack output files with an interactive zoomable treemap.
- `--webpack-copy-php` – enables copying all PHP files from the source directory (default is `src`) and its subfolders to the output directory.
- `--webpack-devtool` – controls how source maps are generated. See options at <https://webpack.js.org/configuration/devtool/#devtool>.
- `--webpack-no-externals` – disables scripts’ assets generation, and omits the list of default externals.
- `--webpack-src-dir` – Allows customization of the source code directory. Default is `src`.
- `--output-path` – Allows customization of the output directory. Default is `build`.

Experimental support for the `block.json` `viewModule` field is available via the `--experimental-modules` option. With this option enabled, script and module fields will all be compiled. The `viewModule` field is analogous to the `viewScript` field, but will compile a module and should be registered in WordPress using the Modules API.

Advanced information

This script uses [webpack](#) behind the scenes. It’ll look for a webpack config in the top-level directory of your package and will use it if it finds one. If none is found, it’ll use the default config provided by `@wordpress/scripts` packages. Learn more in the [Advanced Usage](#) section.

test-e2e

Launches the End-To-End (E2E) test runner. Writing tests can be done using the [Jest API](#) in combination with the [Puppeteer API](#):

[Jest](#) is a delightful JavaScript Testing Framework with a focus on simplicity.

[Puppeteer](#) is a Node library which provides a high-level API to control Chrome or Chromium over the [DevTools Protocol](#). Puppeteer runs [headless](#) by default, but can be configured to run full (non-headless) Chrome or Chromium.

Example:

```
{  
  "scripts": {  
    "test:e2e": "wp-scripts test-e2e",  
    "test:e2e:help": "wp-scripts test-e2e --help",  
    "test:e2e:debug": "wp-scripts --inspect-brk test-e2e --puppeteer-d  
  }  
}
```

This is how you execute those scripts using the presented setup:

- `npm run test:e2e` – runs all e2e tests.
- `npm run test:e2e:help` – prints all available options to configure e2e test runner.
- `npm run test:e2e -- --puppeteer-interactive` – runs all e2e tests interactively.
- `npm run test:e2e FILE_NAME -- --puppeteer-interactive` – runs one test file interactively.
- `npm run test:e2e:watch -- --puppeteer-interactive` – runs all tests interactively and watch for changes.
- `npm run test:e2e:debug` – runs all tests interactively and enables [debugging tests](#).

Jest will look for test files with any of the following popular naming conventions:

- Files with `.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) suffix at any level of depth in `spec` folders.
- Files with `.spec.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) suffix.

This script automatically detects the best config to start Puppeteer but sometimes you may need to specify custom options:

- You can add a `jest-puppeteer.config.js` at the root of the project or define a custom path using `JEST_PUPPETEER_CONFIG` environment variable. Check [jest-puppeteer](#) for more details.

We enforce that all tests run serially in the current process using `--runInBand` Jest CLI option to avoid conflicts between tests caused by the fact that they share the same WordPress instance.

Failed Test Artifacts

When tests fail, both a screenshot and an HTML snapshot will be taken of the page and stored in the `artifacts` / directory at the root of your project. These snapshots may help debug failed tests during development or when running tests in a CI environment.

The `artifacts/` directory can be customized by setting the `WP_ARTIFACTS_PATH` environment variable to the relative path of the desired directory within your project's root. For example: to change the default directory from `artifacts/` to `my/custom/artifacts`, you could use `WP_ARTIFACTS_PATH=my/custom/artifacts npm run test:e2e`.

Advanced information

It uses [Jest](#) behind the scenes and you are able to use all of its [CLI options](#). You can also run `./node_modules/.bin/wp-scripts test:e2e --help` or `npm run test:e2e:help` (as mentioned above) to view all of the available options. Learn more in the [Advanced Usage](#) section.

Should there be any situation where you want to provide your own Jest config, you can do so.

- the command receives a `--config` argument. Example: `wp-scripts test-e2e --config my-jest-config.js`.
- there is a file called `jest-e2e.config.js`, `jest-e2e.config.json`, `jest.config.js`, or `jest.config.json` in the top-level directory of your package (at the same level than your `package.json`).
- a `jest` object can be provided in the `package.json` file with the test configuration.

[test-unit-js](#)

Alias: `test-unit-jest`

Launches the unit test runner. Writing tests can be done using the [Jest API](#).

Example:

```
{  
  "scripts": {  
    "test:unit": "wp-scripts test-unit-js",  
    "test:unit:help": "wp-scripts test-unit-js --help",  
    "test:unit:watch": "wp-scripts test-unit-js --watch",  
    "test:unit:debug": "wp-scripts --inspect-brk test-unit-js --runInB  
  }  
}
```

This is how you execute those scripts using the presented setup:

- `npm run test:unit` – runs all unit tests.
- `npm run test:unit:help` – prints all available options to configure unit tests runner.
- `npm run test:unit:watch` – runs all unit tests in the watch mode.
- `npm run test:unit:debug` – runs all unit tests in [debug mode](#).

Jest will look for test files with any of the following popular naming conventions:

- Files with `.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) suffix located at any level of depth in `__tests__` folders.
- Files with `.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) suffix directly located in `test` folders.
- Files with `.test.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) suffix.

Advanced information

It uses [Jest](#) behind the scenes and you are able to use all of its [CLI options](#). You can also run `./node_modules/.bin/wp-scripts test:unit --help` or `npm run test:unit:help` (as mentioned above) to view all of the available options. By default, it uses the set of recommended options defined in [@wordpress/jest-preset-default](#) npm package. You can override them with your own options as described in [Jest documentation](#). Learn more in the [Advanced Usage](#) section.

Should there be any situation where you want to provide your own Jest config, you can do so.

- the command receives a `--config` argument. Example: `wp-scripts test-unit --config my-jest-config.js`.
- there is a file called `jest-unit.config.js`, `jest-unit.config.json`, `jest.config.js`, or `jest.config.json` in the top-level directory of your package (at the same level than your `package.json`).
- a `jest` object can be provided in the `package.json` file with the test configuration.

[test-playwright](#)

Launches the Playwright End-To-End (E2E) test runner. Similar to Puppeteer, it provides a high-level API to control a headless browser.

Refer to the [Getting Started guide](#) to learn how to write tests.

Example:

```
{  
  "scripts": {  
    "test:playwright": "wp-scripts test-playwright",  
    "test:playwright:help": "wp-scripts test-playwright --help",  
    "test:playwright:debug": "wp-scripts test-playwright --debug"  
  }  
}
```

This is how you execute those scripts using the presented setup:

- `npm run test:playwright` – runs all tests.
- `npm run test:playwright:help` – prints all available options to configure the test runner.
- `npm run test:playwright:debug` – runs all tests interactively with the Playwright inspector.
- `npm run test:playwright FILE_NAME` – runs a specific test file.
- `npm run test:playwright --watch` – runs all tests interactively with watch mode and enhanced debugging.

By default, Playwright looks for JavaScript or TypeScript files with `.test` or `.spec` suffix in the project root-level `/specs` folder, for example `/specs/login-screen.wrong-credentials.spec.ts`.

This script automatically detects the best config to start Playwright, but sometimes you may need to specify custom options.

To do so, you can add a file called `playwright.config.ts` or

`playwright.config.js` in the top-level directory of your package (at the same level as your `package.json`).

Failed Test Artifacts

When tests fail, snapshots will be taken of the page and stored in the `artifacts/` directory at the root of your project. These snapshots may help debug failed tests during development or when running tests in a CI environment.

The `artifacts/` directory can be customized by setting the `WP_ARTIFACTS_PATH` environment variable to the relative path of the desired directory within your project's root. For example: to change the default directory from `artifacts/` to `my/custom/artifacts`, you could use `WP_ARTIFACTS_PATH=my/custom/artifacts npm run test:playwright`.

Advanced information

You are able to use all of Playwright's [CLI options](#). You can also run `./node_modules/.bin/wp-scripts test-playwright --help` or `npm run test:playwright:help` (as mentioned above) to view all the available options. Learn more in the [Advanced Usage](#) section.

[Passing Node.js options](#)

`wp-scripts` supports the full array of [Node.js CLI options](#). They can be passed after the `wp-scripts` command and before the script name.

```
wp-scripts [NODE_OPTIONS] script
```

[Debugging tests](#)

One common use-case for passing Node.js options is debugging your tests.

Tests can be debugged by any [inspector client](#) that supports the [Chrome DevTools Protocol](#).

Follow the instructions for debugging Node.js with your favorite supported browser or IDE. When the instructions say to use `node --inspect script.js` or `node --inspect-brk script.js`, simply use `wp-scripts --inspect script` or `wp-scripts --inspect-brk script` instead.

Google Chrome and Visual Studio Code are used as examples below.

Debugging in Google Chrome

Place `debugger;` statements in any test and run `wp-scripts --inspect-brk test-unit.js --runInBand --no-cache` (or `npm run test:unit:debug` from above).

Then open `about:inspect` in Google Chrome and select `inspect` on your process.

A breakpoint will be set at the first line of the script (this is done to give you time to open the developer tools and to prevent Jest from executing before you have time to do so). Click the resume button in the upper right panel of the dev tools to continue execution. When Jest executes

the test that contains the debugger statement, execution will pause and you can examine the current scope and call stack.

Debugging in Visual Studio Code

Debugging npm scripts is supported out of the box for Visual Studio Code as of [version 1.23](#) and can be used to debug Jest unit tests.

Make sure `wp-scripts --inspect-brk test-unit-js --runInBand --no-cache` is saved as `test:unit:debug` in your `package.json` file to run tests in Visual Studio Code.

When debugging, set a breakpoint in your tests by clicking on a line in the editor's left margin by the line numbers.

Then open npm scripts in the explorer or run **Explorer: Focus on NPM Scripts View** in the command palette to see the npm scripts. To start the tests, click the debug icon next to `test:unit:debug`.

The tests will start running, and execution will pause on your selected line so you can inspect the current scope and call stack within the editor.

See [Debugging in Visual Studio Code](#) for more details on using the Visual Studio Code debugger.

Debugging e2e tests

Since e2e tests run both in the node context *and* the (usually headless) browser context, not all lines of code can have breakpoints set within the inspector client—only the node context is debugged in the inspector client.

The code executed in the node context includes all of the test files *excluding* code within `page.evaluate` functions. The `page.evaluate` functions and the rest of your app code is executed within the browser context.

Test code (node context) can be debugged normally using the instructions above.

To also debug the browser context, run `wp-scripts --inspect-brk test-e2e --puppeteer-devtools`. The `--puppeteer-devtools` option (or the `PUPPETEER_DEVTOOLS="true"` environment variable when used with `PUPPETEER_HEADLESS="false"`) will disable headless mode and launch the browser with the devtools already open. Breakpoints can then be set in the browser context using these devtools.

For more e2e debugging tips check out the [Puppeteer debugging docs](#).

Advanced Usage

In general, this package should be used with the set of recommended config files. While it's possible to override every single config file provided, if you have to do it, it means that your use case is far more complicated than anticipated. If that happens, it would be better to avoid using the whole abstraction layer and set up your project with full control over tooling used.

Working with build scripts

The `build` and `start` commands use [webpack](#) behind the scenes. webpack is a tool that helps you transform your code into something else. For example: it can take code written in ESNext and output ES5 compatible code that is minified for production.

Default webpack config

`@wordpress/scripts` bundles the default webpack config used as a base by the WordPress editor. These are the defaults:

- [Entry](#): the entry points for your project get detected by scanning all script fields in `block.json` files located in the `src` directory. The fallback entry point is `src/index.js` (other supported extensions: `.jsx`, `.ts`, and `.tsx`) in case there is no `block.json` file found.
- [Output](#): `build/[name].js`, for example: `build/index.js`, or `build/my-block/index.js`.
- [Loaders](#):
 - [babel-loader](#) allows transpiling JavaScript and TypeScript files using Babel and webpack.
 - [@svgr/webpack](#) and [url-loader](#) makes it possible to handle SVG files in JavaScript code.
 - [css-loader](#) chained with [postcss-loader](#) and [sass-loader](#) let webpack process CSS, SASS or SCSS files referenced in JavaScript files.
- [Plugins](#) (among others):
 - [CopyWebpackPlugin](#) copies all `block.json` files discovered in the `src` directory to the build directory.
 - [MiniCssExtractPlugin](#) extracts CSS into separate files. It creates a CSS file per JavaScript entry point which contains CSS.
 - [@wordpress/dependency-extraction-webpack-plugin](#) is used with the default configuration to ensure that WordPress provided scripts are not included in the built bundle.

Using CSS

Example:

```
// index.scss
$body-color: red;

.wp-block-my-block {
    color: $body-color;
}

/* style.css */
.wp-block-my-block {
    background-color: black;
}

// index.js
import './index.pcss';
import './index.scss';
import './style.css';
```

When you run the build using the default command `wp-scripts build` (also applies to `start`) in addition to the JavaScript file `index.js` generated in the build folder, you should see two more files:

1. `index.css` – all imported CSS files are bundled into one chunk named after the entry point, which defaults to `index.js`, and thus the file created becomes `index.css`. This is for styles used only in the editor.
2. `style-index.css` – imported `style.css` file(s) (applies to PCSS, SASS and SCSS extensions) get bundled into one `style-index.css` file that is meant to be used both on the front-end and in the editor.

You can also have multiple entry points as described in the docs for the script:

```
wp-scripts start entry-one.js entry-two.js --output-path=custom
```

If you do so, then CSS files generated will follow the names of the entry points: `entry-one.css` and `entry-two.css`.

Avoid using `style` keyword in an entry point name, this might break your build process.

You can also bundle CSS modules by prefixing `.module` to the extension, e.g. `style.module.scss`. Otherwise, these files are handled like all other `style.scss`. They will also be extracted into `style-index.css`.

Using fonts and images

It is possible to reference font (`woff`, `woff2`, `eot`, `ttf` and `otf`) and image (`bmp`, `png`, `jpg`, `jpeg`, `gif` and `webp`) files from CSS that is controlled by webpack as explained in the previous section.

Example:

```
/* style.css */
@font-face {
    font-family: Gilbert;
    src: url( ../assets/gilbert-color.otf );
}
.wp-block-my-block {
    background-color: url( ../assets/block-background.png );
    font-family: Gilbert;
}
```

Using SVG

Example:

```
import starUrl, { ReactComponent as Star } from './star.svg';

const App = () => (
    <div>
        <img src={ starUrl } alt="star" />
        <Star />
    </div>
);
```

Provide your own webpack config

Should there be any situation where you want to provide your own webpack config, you can do so. The `build` and `start` commands will use your provided file when:

- the command receives a `--config` argument. Example: `wp-scripts build --config my-own-webpack-config.js`.
- there is a file called `webpack.config.js` or `webpack.config.babel.js` in the top-level directory of your project (at the same level as `package.json`).

Extending the webpack config

To extend the provided webpack config, or replace subsections within the provided webpack config, you can provide your own `webpack.config.js` file, require the provided `webpack.config.js` file, and use the [spread operator](#) to import all of or part of the provided configuration.

In the example below, a `webpack.config.js` file is added to the root folder extending the provided webpack config to include custom logic to parse module's source and convert it to a JavaScript object using [toml](#). It may be useful to import toml or other non-JSON files as JSON, without specific loaders:

```
const toml = require('toml');
const defaultConfig = require('@wordpress/scripts/config/webpack.config')

module.exports = {
    ...defaultConfig,
    module: {
        ...defaultConfig.module,
        rules: [
            ...defaultConfig.module.rules,
            {
                test: /\.toml$/,
                type: 'json',
                parser: {
                    parse: toml.parse,
                },
            },
        ],
    },
};
```

If you follow this approach, please, be aware that:

- You should keep using the `wp-scripts` commands (`start` and `build`). Do not use `webpack` directly.
- Future versions of this package may change what webpack and Babel plugins we bundle, default configs, etc. Should those changes be necessary, they will be registered in the [package's CHANGELOG](#), so make sure to read it before upgrading.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/scripts”](#)

[Previous @wordpress/router](#) [Previous: @wordpress/router](#)

[Next @wordpress/server-side-render](#) [Next: @wordpress/server-side-render](#)

@wordpress/server-side-render

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Props](#)
 - [attributes](#)
 - [block](#)
 - [className](#)
 - [httpMethod](#)
 - [skipBlockSupportAttributes](#)
 - [urlQueryArgs](#)
 - [EmptyResponsePlaceholder](#)
 - [ErrorResponsePlaceholder](#)
 - [LoadingResponsePlaceholder](#)
- [Usage](#)
- [Output](#)
- [API Endpoint](#)
- [Contributing to this package](#)

[↑ Back to top](#)

ServerSideRender is a component used for server-side rendering a preview of dynamic blocks to display in the editor. Server-side rendering in a block's `edit` function should be limited to blocks that are heavily dependent on existing PHP rendering logic that is heavily intertwined with data, particularly when there are no endpoints available.

ServerSideRender may also be used when a legacy block is provided as a backward compatibility measure, rather than needing to re-write the deprecated code that the block may depend on.

ServerSideRender should be regarded as a fallback or legacy mechanism, it is not appropriate for developing new features against.

New blocks should be built in conjunction with any necessary REST API endpoints, so that JavaScript can be used for rendering client-side in the `edit` function. This gives the best user experience, instead of relying on using the PHP `render_callback`. The logic necessary for rendering should be included in the endpoint, so that both the client-side JavaScript and server-side PHP logic should require a minimal amount of differences.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

Installation

Install the module

```
npm install @wordpress/server-side-render --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.*

Usage

The props accepted by the component are described below.

Props

attributes

An object containing the attributes of the block to be server-side rendered.

E.g: { `displayAsDropdown: true` }, { `showHierarchy: true` }, etc...

- Type: `Object`
- Required: No

block

The identifier of the block to be server-side rendered.

Examples: "core/archives", "core/latest-comments", "core/rss", etc...

- Type: `String`
- Required: Yes

className

A class added to the DOM element that wraps the server side rendered block.
Examples: “my-custom-server-side-rendered”.

- Type: String
- Required: No

httpMethod

The HTTP request method to use, either ‘GET’ or ‘POST’. It’s ‘GET’ by default. The ‘POST’ value will cause an error on WP earlier than 5.5, unless ‘rest_endpoints’ is filtered in PHP to allow this. If ‘POST’, this sends the attributes in the request body, not in the URL. This can allow a bigger attributes object.

- Type: String
- Required: No
- Default: ‘GET’

Example:

```
function add_rest_method( $endpoints ) {
    if ( is_wp_version_compatible( '5.5' ) ) {
        return $endpoints;
    }

    foreach ( $endpoints as $route => $handler ) {
        if ( isset( $endpoints[ $route ][0] ) ) {
            $endpoints[ $route ][0]['methods'] = [ WP_REST_Server::READABLE ];
        }
    }

    return $endpoints;
}
add_filter( 'rest_endpoints', 'add_rest_method');
```

skipBlockSupportAttributes

Remove attributes and style properties applied by the block supports. This prevents duplication of styles in the block wrapper and the ServerSideRender components. Even if certain features skip serialization to HTML markup by `__experimentalSkipSerialization`, all attributes and style properties are removed.

- Type: Boolean
- Required: No
- Default: false

urlQueryArgs

Query arguments to apply to the request URL.
E.g: { post_id: 12 }.

- Type: Object

- Required: No

[EmptyResponsePlaceholder](#)

The component is rendered when the API response is empty. The component will receive the value of the API response, and all props passed into `ServerSideRenderer`.

- Type: Component
- Required: No

[ErrorResponsePlaceholder](#)

The component is rendered when the API response is an error. The component will receive the value of the API response, and all props passed into `ServerSideRenderer`.

- Type: Component
- Required: No

[LoadingResponsePlaceholder](#)

The component is rendered while the API request is being processed (loading state). The component will receive the value of the API response, and all props passed into `ServerSideRenderer`.

- Type: Component
- Required: No

Example usage

```
const MyServerSideRender = () => (
  <ServerSideRender LoadingResponsePlaceholder={ MyAmazingPlaceholder } >
);
```

[Usage](#)

Render core/archives preview.

```
import ServerSideRender from '@wordpress/server-side-render';

const MyServerSideRender = () => (
  <ServerSideRender
    block="core/archives"
    attributes={ {
      showPostCounts: true,
      displayAsDropdown: false,
    } }
  />
);
```

If imported from the `wp` global, an alias is required to work in JSX.

```
const { serverSideRender: ServerSideRender } = wp;
```

```
const MyServerSideRender = () => (
  <ServerSideRender
    block="core/archives"
    attributes={ {
      showPostCounts: true,
      displayAsDropdown: false,
    } }
  />
);
```

Output

Output uses the block's `render_callback` function, set when defining the block.

API Endpoint

The API endpoint for getting the output for ServerSideRender is `/wp/v2/block-renderer/:block`. It will use the block's `render_callback` method.

If you pass `attributes` to `ServerSideRender`, the block must also be registered and have its attributes defined in PHP.

```
register_block_type(
  'core/archives',
  array(
    'api_version' => 3,
    'attributes' => array(
      'showPostCounts' => array(
        'type' => 'boolean',
        'default' => false,
      ),
      'displayAsDropdown' => array(
        'type' => 'boolean',
        'default' => false,
      ),
    ),
    'render_callback' => 'render_block_core_archives',
  )
);
```

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/server-side-render”](#)

[Previous @wordpress/scripts](#) [Previous: @wordpress/scripts](#)
[Next @wordpress/shortcode](#) [Next: @wordpress/shortcode](#)

@wordpress/shortcode

In this article

[Table of Contents](#)

- [Installation](#)
- [API](#)
 - [attrs](#)
 - [default](#)
 - [fromMatch](#)
 - [next](#)
 - [regexp](#)
 - [replace](#)
 - [string](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Shortcode module for WordPress.

[Installation](#)

Install the module

```
npm install @wordpress/shortcode --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](#)

[attrs](#)

Parse shortcode attributes.

Shortcodes accept many types of attributes. These can chiefly be divided into named and numeric attributes:

Named attributes are assigned on a key/value basis, while numeric attributes are treated as an array.

Named attributes can be formatted as either `name="value"`, `name='value'`, or `name=value`. Numeric attributes can be formatted as `"value"` or just `value`.

Parameters

- `text string`: Serialised shortcode attributes.

Returns

- `WPShortcodeAttrs`: Parsed shortcode attributes.

default

Creates a shortcode instance.

To access a raw representation of a shortcode, pass an `options` object, containing a `tag` string, a string or object of `attrs`, a string indicating the type of the shortcode ('single', 'self-closing', or 'closed'), and a `content` string.

Parameters

- `options Object`: Options as described.

Returns

- `WPShortcode`: Shortcode instance.

fromMatch

Generate a Shortcode Object from a RegExp match.

Accepts a `match` object from calling `regexp.exec()` on a RegExp generated by `regexp().match` can also be set to the arguments from a callback passed to `regexp.replace()`.

Parameters

- `match Array`: Match array.

Returns

- `WPShortcode`: Shortcode instance.

next

Find the next matching shortcode.

Parameters

- *tag string*: Shortcode tag.
- *text string*: Text to search.
- *index number*: Index to start search from.

Returns

- `WPShortcodeMatch` | `undefined`: Matched information.

regexp

Generate a RegExp to identify a shortcode.

The base regex is functionally equivalent to the one found in `get_shortcode_regex()` in `wp-includes/shortcodes.php`.

Capture groups:

1. An extra [to allow for escaping shortcodes with double []]
2. The shortcode name
3. The shortcode argument list
4. The self closing /
5. The content of a shortcode when it wraps some content.
6. The closing tag.
7. An extra] to allow for escaping shortcodes with double []]

Parameters

- *tag string*: Shortcode tag.

Returns

- `RegExp`: Shortcode RegExp.

replace

Replace matching shortcodes in a block of text.

Parameters

- *tag string*: Shortcode tag.
- *text string*: Text to search.
- *callback Function*: Function to process the match and return replacement string.

Returns

- `string`: Text with shortcodes replaced.

string

Generate a string from shortcode parameters.

Creates a shortcode instance and returns a string.

Accepts the same `options` as the `shortcode()` constructor, containing a `tag` string, a string or object of `attrs`, a boolean indicating whether to format the shortcode using a `single` tag, and a `content` string.

Parameters

- `options` Object:

Returns

- `string`: String representation of the shortcode.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/shortcode](#)

[Previous @wordpress/server-side-render](#) Previous: [@wordpress/server-side-render](#)
[Next @wordpress/style-engine](#) Next: [@wordpress/style-engine](#)

@wordpress/style-engine

In this article

Table of Contents

- [Please note](#)
- [Backend API](#)
 - [wp_style_engine_get_styles\(\)](#)
 - [wp_style_engine_get_stylesheet_from_css_rules\(\)](#)
 - [wp_style_engine_get_stylesheet_from_context\(\)](#)
- [Installation \(JS only\)](#)
- [Usage](#)
 - [compileCSS](#)
 - [getCSSRules](#)

- [Glossary](#)

[↑ Back to top](#)

The Style Engine aims to provide a consistent API for rendering styling for blocks across both client-side and server-side applications.

Initially, it will offer a single, centralized agent responsible for generating block styles, and, in later phases, it will also assume the responsibility of processing and rendering optimized frontend CSS.

Please note

This package is new as of WordPress 6.1 and therefore in its infancy.

Upcoming tasks on the roadmap include, but are not limited to, the following:

- Consolidate global and block style rendering and enqueueing (ongoing)
- Explore pre-render CSS rule processing with the intention of deduplicating other common and/or repetitive block styles. (ongoing)
- Extend the scope of semantic class names and/or design token expression, and encapsulate rules into stable utility classes.
- Explore pre-render CSS rule processing with the intention of deduplicating other common and/or repetitive block styles.
- Propose a way to control hierarchy and specificity, and make the style hierarchy cascade accessible and predictable. This might include preparing for CSS cascade layers until they become more widely supported, and allowing for opt-in support in Gutenberg via theme.json.
- Refactor all blocks to consistently use the “style” attribute for all customizations, that is, deprecate preset-specific attributes such as `attributes.fontSize`.

For more information about the roadmap, please refer to [Block editor styles: initiatives and goals](#) and the [Github project board](#).

If you’re making changes or additions to the Style Engine, please take a moment to read the [notes on contributing](#).

Backend API

`wp_style_engine_get_styles()`

Global public function to generate styles from a single style object, e.g., the value of a [block’s attributes.style object](#) or the [top level styles in theme.json](#).

See also [Using the Style Engine to generate block supports styles](#).

Parameters

- `$block_styles array` A block’s `attributes.style` object or the top level styles in theme.json

- `$options array<string|boolean>` An array of options to determine the output.
 - `context string` An identifier describing the origin of the style object, e.g., ‘block-supports’ or ‘global-styles’. Default is ‘block-supports’. When both `context` and `selector` are set, the Style Engine will store the CSS rules using the `context` as a key.
 - `convert_vars_to_classnames boolean` Whether to skip converting CSS var:? values to var(–wp–preset–*) values. Default is `false`.
 - `selector string` When a selector is passed, `generate()` will return a full CSS rule `$selector { ...rules }`, otherwise a concatenated string of properties and values.

Returns

`array<string|array>|null`

```
array(
  'css'          => (string) A CSS ruleset or declarations block format
  'declarations' => (array) An array of property/value pairs representing
  'classnames'   => (string) Classnames separated by a space.
);
```

It will return compiled CSS declarations for inline styles, or, where a selector is provided, a complete CSS rule.

To enqueue a style for rendering in the site’s frontend, the `$options` array requires the following:

1. **selector (string)** – this is the CSS selector for your block style CSS declarations.
2. **context (string)** – this tells the Style Engine where to store the styles. Styles in the same context will be stored together.

`wp_style_engine_get_styles` will return the compiled CSS and CSS declarations array.

Usage

As mentioned, `wp_style_engine_get_styles()` is useful whenever you wish to generate CSS and/or classnames from a **block’s style object**. A good example is [using the Style Engine to generate block supports styles](#).

In the following snippet, we’re taking the style object from a block’s attributes and passing it to the Style Engine to get the styles. By passing a `context` in the options, the Style Engine will store the styles for later retrieval, for example, should you wish to batch enqueue a set of CSS rules.

```
$block_attributes = array(
  'style' => array(
    'spacing' => array( 'padding' => '100px' ),
  ),
);

$styles = wp_style_engine_get_styles(
  $block_attributes['style'],
  array(
    'selector' => '.a-selector',
    'context'   => 'block-supports',
```

```

    )
);

print_r( $styles );

/*
array(
    'css'          => '.a-selector{padding:100px}'
    'declarations' => array( 'padding' => '100px' )
)
*/

```

[wp_style_engine_get_stylesheet_from_css_rules\(\)](#)

Use this function to compile and return a stylesheet for any CSS rules. The Style Engine will automatically merge declarations and combine selectors.

This function acts as a CSS compiler, but will also register the styles in a store where a `context` string is passed in the options.

Parameters

- `$css_rules array<array>`
- `$options array<string|bool>` An array of options to determine the output.
 - `context` string An identifier describing the origin of the style object, e.g., ‘block-supports’ or ‘global-styles’. Default is ‘block-supports’. When set, the Style Engine will attempt to store the CSS rules.
 - `prettyify` bool Whether to add new lines and indents to output. Default is to inherit the value of the global constant `SCRIPT_DEBUG`, if it is defined.
 - `optimize` bool Whether to optimize the CSS output, e.g., combine rules. Default is `false`.

Returns

`string` A compiled CSS string based on `$css_rules`.

Usage

Useful for when you wish to compile a bespoke set of CSS rules from a series of selector + declaration items.

The Style Engine will return a sanitized stylesheet. By passing a `context` identifier in the options, the Style Engine will store the styles for later retrieval, for example, should you wish to batch enqueue a set of CSS rules.

You can call `wp_style_engine_get_stylesheet_from_css_rules()` multiple times, and, so long as your styles use the same `context` identifier, they will be stored together.

```

$styles = array(
    array(
        'selector'      => '.wp-pumpkin',
        'declarations' => array( 'color' => 'orange' )
    ),
    array(
        'selector'      => '.wp-tomato',
        'declarations' => array( 'color' => 'red' )
)

```

```

),
array(
    'selector'      => '.wp-tomato',
    'declarations' => array( 'padding' => '100px' )
),
array(
    'selector'      => '.wp-kumquat',
    'declarations' => array( 'color' => 'orange' )
),
);
$stylesheet = wp_style_engine_get_stylesheet_from_css_rules(
    $styles,
    array(
        'context' => 'block-supports', // Indicates that these styles should be merged
    )
);
print_r( $stylesheet ); // .wp-pumpkin,.wp-kumquat{color:orange}.wp-tomato

```

[wp style engine get stylesheet from context\(\)](#)

Returns compiled CSS from a stored context, if found.

Parameters

- *\$store_name* string An identifier describing the origin of the style object, e.g., ‘block-supports’ or ‘global-styles’. Default is ‘block-supports’.
- *\$options* array<bool> An array of options to determine the output.
 - *prettify* bool Whether to add new lines and indents to output. Default is to inherit the value of the global constant `SCRIPT_DEBUG`, if it is defined.
 - *optimize* bool Whether to optimize the CSS output, e.g., combine rules. Default is `false`.

Returns

string A compiled CSS string from the stored CSS rules.

Usage

Use this function to generate a stylesheet using all the styles stored under a specific context identifier.

A use case would be when you wish to enqueue all stored styles for rendering to the frontend. The Style Engine will merge and deduplicate styles upon retrieval.

```

// First, let's gather and register our styles.
$styles = array(
    array(
        'selector'      => '.wp-apple',
        'declarations' => array( 'color' => 'green' )
    ),
);
wp_style_engine_get_stylesheet_from_css_rules(
    $styles,

```

```

array(
    'context' => 'fruit-styles',
)
);

// Later, we fetch compiled rules from context store.
$stylesheet = wp_style_engine_get_stylesheet_from_context( 'fruit-styles' )

print_r( $stylesheet ); // .wp-apple{color:green;}

if ( ! empty( $stylesheet ) ) {
    wp_register_style( 'my-stylesheet', false, array(), true, true );
    wp_add_inline_style( 'my-stylesheet', $stylesheet );
    wp_enqueue_style( 'my-stylesheet' );
}

```

Installation (JS only)

Install the module

```
npm install @wordpress/style-engine --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.*

Usage

compileCSS

Generates a stylesheet for a given style object and selector.

Parameters

- **style Style**: Style object, for example, the value of a block's attributes.style object or the top level styles in theme.json
- **options StyleOptions**: Options object with settings to adjust how the styles are generated.

Returns

- **string**: A generated stylesheet or inline style declarations.

Changelog

6.1.0 Introduced in WordPress core.

getCSSRules

Returns a JSON representation of the generated CSS rules.

Parameters

- **style Style**: Style object, for example, the value of a block's attributes.style object or the top level styles in theme.json
- **options StyleOptions**: Options object with settings to adjust how the styles are generated.

Returns

- **GeneratedCSSRule []**: A collection of objects containing the selector, if any, the CSS property key (camelcase) and parsed CSS value.

Changelog

6.1.0 Introduced in WordPress core.

Glossary

A guide to the terms and variable names referenced by the Style Engine package.

Block style (Gutenberg internal)

An object comprising a block's style attribute that contains a block's style values. E.g.,
`{ spacing: { margin: '10px' }, color: { ... }, ... }`

Context

An identifier for a group of styles that share a common origin or purpose, e.g., ‘block-supports’ or ‘global-styles’. The context is also used as a key to fetch CSS rules from the store.

CSS declaration or (CSS property declaration)

A CSS property paired with a CSS value. E.g., `color: pink`

CSS declarations block

A set of CSS declarations usually paired with a CSS selector to create a CSS rule.

CSS property

Identifiers that describe stylistic, modifiable features of an HTML element. E.g., `border`, `font-size`, `width...`

CSS rule

A CSS selector followed by a CSS declarations block inside a set of curly braces. Usually found in a CSS stylesheet.

CSS selector (or CSS class selector)

The first component of a CSS rule, a CSS selector is a pattern of elements, classnames or other terms that define the element to which the rule's CSS definitions apply. E.g., `p.my-cool-classname > span`. A CSS selector matches HTML elements based on the contents of the “class” attribute. See [MDN CSS selectors article](#).

CSS stylesheet

A collection of CSS rules contained within a file or within an [HTML style tag](#).

CSS value

The value of a CSS property. The value determines how the property is modified. E.g., the `10vw` in `height: 10vw`.

CSS variables (vars) or CSS custom properties

Properties, whose values can be reused in other CSS declarations. Set using custom property notation (e.g., `--wp--preset--olive: #808000;`) and accessed using the `var()` function (e.g., `color: var(--wp--preset--olive);`). See [MDN article on CSS custom properties](#).

Global styles (Gutenberg internal)

A merged block styles object containing values from a theme's theme.json and user styles settings.

Inline styles

Inline styles are CSS declarations that affect a single HTML element, contained within a style attribute

Processor

Performs compilation and optimization on stored CSS rules. See the class '[WP_Style_Engine_Processor](#)'.

Store

A data object that contains related styles. See the class '[WP_Style_Engine_CSS_Rules_Store](#)'.

First published

February 15, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/style-engine”](#)

[Previous @wordpress/shortcode](#) [Previous: @wordpress/shortcode](#)

[Next @wordpress/style-engine](#) [Using the Style Engine to generate block supports styles](#) [Next: @wordpress/style-engine](#) [Using the Style Engine to generate block supports styles](#)

@wordpress/style-engine Using the Style Engine to generate block supports styles

In this article

Table of Contents

- [Use case](#)
- [Checking for block support and skip serialization](#)
- [Generatingclassnames and CSS custom selectors from presets](#)

[↑ Back to top](#)

[Block supports](#) is the API that allows a block to declare support for certain features.

Where a block declares support for a specific style group or property, e.g., “spacing” or “spacing.padding”, the block’s attributes are extended to include a **style object**.

For example:

```
{  
  "attributes": {  
    "style": {
```

```
        "spacing": {
            "margin": {
                "top": "10px"
            },
            "padding": "1em"
        },
        "typography": {
            "fontSize": "2.2rem"
        }
    }
}
```

Using this object, the Style Engine can generate the classes and CSS required to style the block element.

The global function `wp_style_engine_get_styles` accepts a style object as its first argument, and will output compiled CSS and an array of CSS declaration property/value pairs.

```
$block_styles = array(
    'spacing' => array( 'padding' => '10px', 'margin' => array( 'top' => '1em' ),
    'typography' => array( 'fontSize' => '2.2rem' ),
);
$styles = wp_style_engine_get_styles(
    $block_styles
);
print_r( $styles );
/*
array(
    'css'          => 'padding:10px;margin-top:1em;font-size:2.2rem',
    'declarations' => array( 'padding' => '10px', 'margin-top' => '1em',
)
*/
```

Use case

When [registering a block support](#), it is possible to pass an ‘apply’ callback in the block support config array to add or extend block support attributes with “class” or “style” properties.

If a block has opted into the block support, the values of “class” and “style” will be applied to the block element’s “class” and “style” attributes accordingly when rendered in the frontend HTML. Note, this applies only to server-side rendered blocks, for example, the [Site Title block](#).

The callback receives `$block_type` and `$block_attributes` as arguments. The `style` attribute within `$block_attributes` only contains the raw style object, if any styles have been set for the block, and not any CSS or classnames to be applied to the block's HTML elements.

Here is where `wp_style_engine_get_styles` comes in handy: it will generate CSS and, if appropriate, classnames to be added to the “style” and “class” HTML attributes in the final rendered block markup.

Here is a *very* simplified version of how the [color block support](#) works:

```

function gutenberg_apply_colors_support( $block_type, $block_attributes )
    // Get the color styles from the style object.
    $block_color_styles = isset( $block_attributes['style']['color'] ) ? $block_attributes['style']['color'] : null;

    // Since we only want the color styles, pass the color styles only to
    // $styles = wp_style_engine_get_styles( array( 'color' => $block_color_styles ) );
    // Return the generated styles to be applied to the block's HTML element.
    return array(
        'style' => $styles['css'],
        'class' => $styles['classnames']
    );
}

// Register the block support.
WP_Block_Supports::get_instance()->register(
    'colors',
    array(
        'register_attribute' => 'gutenberg_register_colors_support',
        'apply'                => 'gutenberg_apply_colors_support',
    )
);

```

It's important to note that, for now, the Style Engine will only generate styles for the following, core block supports:

- border
- color
- spacing
- typography

In future releases, it will be possible to extend this list.

Checking for block support and skip serialization

Before passing the block style object to the Style Engine, you'll need to take into account:

1. whether the theme has elected to support a particular block style, and
2. whether a block has elected to “skip serialization” of that particular block style, that is, opt-out of automatic application of styles to the block's element (usually in order to do it via the block's internals). See the [block API documentation](#) for further information.

If a block either:

- has no support for a style, or
- skips serialization of that style

it's likely that you'll want to remove those style values from the style object before passing it to the Style Engine with help of two functions:

- `wp_should_skip_block_supports_serialization()`
- [block_has_support\(\)](#)

We can now update the “apply” callback code above so that we’ll only return “style” and “class”, where a block has support, and it doesn’t skip serialization:

```
function gutenberg_apply_colors_support( $block_type, $block_attributes )  
    // The return value.  
    $attributes = array();  
  
    // Return early if the block skips all serialization for block support  
    if ( gutenberg_should_skip_block_supports_serialization( $block_type,  
        return $attributes;  
    }  
  
    // Checks for support and skip serialization.  
    $has_text_support = block_has_support( $block_type );  
    $has_background_support = block_has_support( $block_type );  
    $skips_serialization_of_color_text = wp_should_skip_block_supports_serialization( $block_type, 'color' );  
    $skips_serialization_of_color_background = wp_should_skip_block_supports_serialization( $block_type, 'background' );  
  
    // Get the color styles from the style object.  
    $block_color_styles = isset( $block_attributes['style']['color'] ) ? $block_attributes['style']['color'] : null;  
  
    // The mutated styles object we're going to pass to wp_style_engine_get_styles()  
    $color_block_styles = array();  
  
    // Set the color style values according to whether the block has support  
    $spacing_block_styles['text'] = null;  
    $spacing_block_styles['background'] = null;  
    if ( $has_text_support && ! $skips_serialization_of_color_text ) {  
        $spacing_block_styles['text'] = $block_color_styles['text'] ?? null;  
    }  
    if $has_background_support && ! $skips_serialization_of_color_background {  
        $spacing_block_styles['background'] = $block_color_styles['background'] ?? null;  
    }  
  
    // Pass the color styles, excluding those that have no support or skip serialization  
    $styles = wp_style_engine_get_styles( array( 'color' => $block_color_styles ) );  
  
    // Return the generated styles to be applied to the block's HTML element  
    return array(  
        'style' => $styles['css'],  
        'class' => $styles['classnames']  
    );  
}
```

Generating classnames and CSS custom selectors from presets

Many of theme.json’s presets will generate both CSS custom properties and CSS rules (consisting of a selector and the CSS declarations) on the frontend.

Styling a block using these presets normally involves adding the selector to the “className” attribute of the block.

For styles that can have preset values, such as text color and font family, the Style Engine knows how to construct the classnames using the preset slug.

To discern CSS values from preset values, the Style Engine expects a special format.

Preset values must follow the pattern `var:preset|<PRESET_TYPE>|<PRESET_SLUG>`.

When the Style Engine encounters these values, it will parse them and create a CSS value of `var(--wp--preset--font-size--small)` and/or generate a classname if required.

Example:

```
// Let's say the block attributes styles contain a fontSize preset slug of
// $block_attributes['fontSize'] = 'var:preset|font-size|small';
$preset_font_size      = "var:preset|font-size|{$block_attributes['fontSize']}";
// Now let's say the block attributes styles contain a backgroundColor preset
// $block_attributes['backgroundColor'] = 'var:preset|color|blue';
$preset_background_color = "var:preset|color|{$block_attributes['backgroundColor']}";

$block_styles = array(
    'typography' => array( 'fontSize' => $preset_font_size ),
    'color'       => array( 'background' => $preset_background_color ),
    'spacing'     => array( 'padding' => '10px', 'margin' => array( 'top' =>
);

$styles = wp_style_engine_get_styles(
    $block_styles
);
print_r( $styles );

/*
array(
    'css'           => 'background-color:var(--wp--preset--color--blue);padding:0;
    'declarations' => array(
        'background-color' => 'var(--wp--preset--color--blue)',
        'padding'          => '10px',
        'margin-top'       => '1em',
        'font-size'        => 'var(--wp--preset--font-size--small)',
    ),
    'classnames'   => 'has-background has-blue-background-color has-small-padding'
)
*/
```

If you don't want the Style Engine to output the CSS custom vars in the generated CSS string as well, which you might not if you're applying both the CSS rules and classnames to the block element, you can pass `'convert_vars_to_classnames' => true` in the options array.

This flag means “convert the vars to classnames and don't output the vars to the CSS”. The Style Engine will therefore **only** generate the required classnames and omit the CSS custom vars in the CSS.

Using the above example code, observe the different output when we pass the option:

```
$options = array(
    'convert_vars_to_classnames' => true,
```

```
);

$styles = wp_style_engine_get_styles(
    $block_styles,
    $options
);
print_r( $styles );

/*
array(
    'css'          => 'padding:10px;margin-top:1em;',
    'declarations' => array(
        'padding' => '10px',
        'margin-top' => '1em',
    ),
    'classnames'   => 'has-background has-blue-background-color has-small-'
)
*/
*/
```

Read more about [global styles](#) and [preset CSS custom properties](#) and [theme supports](#).

First published

September 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/style-engine Using the Style Engine to generate block supports styles](#)

[Previous @wordpress/style-engine](#) Previous: [@wordpress/style-engine](#)
[Next @wordpress/stylelint-config](#) Next: [@wordpress/stylelint-config](#)

②@wordpress/stylelint-config

In this article

Table of Contents

- [Installation](#)
- [Usage](#)
- [Presets](#)
 - [SCSS](#)
- [Extending the config](#)
- [Contributing to this package](#)

[↑ Back to top](#)

[stylelint](#) configuration rules to ensure your CSS is compliant with the [WordPress CSS Coding Standards](#).

Installation

```
$ npm install @wordpress/stylelint-config --save-dev
```

Note: This package requires Node.js 14.0.0 or later. It is not compatible with older versions.

Usage

If you've installed `@wordpress/stylelint-config` locally within your project, just set your `stylelint` config to:

```
{  
    "extends": "@wordpress/stylelint-config"  
}
```

If you've globally installed `@wordpress/stylelint-config` using the `-g` flag, then you'll need to use the absolute path to `@wordpress/stylelint-config` in your config:

```
{  
    "extends": "/absolute/path/to/@wordpress/stylelint-config"  
}
```

Presets

In addition to the default preset, there is also a SCSS preset. This preset extends both `@wordpress/stylelint-config` and [stylelint-config-recommended-scss](#).

SCSS

```
{  
    "extends": [ "@wordpress/stylelint-config/scss" ]  
}
```

Extending the config

Simply add a "rules" key to your config and add your overrides there.

For example, to change the `indentation` to four spaces and turn off the `number-leading-zero` rule:

```
{  
    "extends": "@wordpress/stylelint-config",  
    "rules": {  
        "indentation": 4,  
        "number-leading-zero": null  
    }  
}
```

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/stylelint-config”](#)

[Previous @wordpress/style-engine Using the Style Engine to generate block supports styles](#)

[Previous: @wordpress/style-engine Using the Style Engine to generate block supports styles](#)

[Next @wordpress/sync](#) [Next: @wordpress/sync](#)

@wordpress/sync

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [connectIndexDb](#)
 - [createSyncProvider](#)
 - [createWebRTCCConnection](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Sync data between multiple peers and persist in a local database.

Installation

Install the module

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

API

[connectIndexDb](#)

Connect function to the IndexedDB persistence provider.

Parameters

- *objectId* Object ID: The object ID.
- *objectType* Object Type: The object type.
- *doc* CRDTDoc: The CRDT document.

Returns

- Promise<() => void>: Promise that resolves when the connection is established.

[createSyncProvider](#)

Create a sync provider.

Parameters

- *connectLocal* ConnectDoc: Connect the document to a local database.
- *connectRemote* ConnectDoc: Connect the document to a remote sync connection.

Returns

- SyncProvider: Sync provider.

[createWebRTCCConnection](#)

Function that creates a new WebRTC Connection.

Parameters

- *config* Object: The object ID.
- *config.signaling* Array<string>:
- *config.password* string:

Returns

- Function: Promise that resolves when the connection is established.

[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](#).

First published

August 9, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress.sync”](#)

[Previous @wordpress/stylelint-config](#) [Previous: @wordpress/stylelint-config](#)

[Next @wordpress/token-list](#) [Next: @wordpress/token-list](#)

@wordpress/token-list

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
- [Browser Support](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Constructable, plain JavaScript [DOMTokenList](#) implementation, supporting non-browser runtimes.

[Installation](#)

Install the module

```
npm install @wordpress/token-list
```

*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.*

[Usage](#)

Construct a new token list, optionally with an initial value. A value with an interface matching [DOMTokenList](#) is returned.

```
import TokenList from '@wordpress/token-list';

const tokens = new TokenList( 'abc def' );
tokens.add( 'ghi' );
tokens.remove( 'def' );
tokens.replace( 'abc', 'xyz' );
console.log( tokens.value );
// "xyz ghi"
```

All [methods of DOMTokenList](#) are implemented.

Note the following implementation divergences from the [specification](#):

- `TokenList#supports` will always return true, regardless of the token passed.
- `TokenList#add` and `TokenList#remove` will simply disregard the empty string argument or whitespace of a token argument, rather than [throwing an error](#).
- An item cannot be referenced by its index as a property. Use `TokenList#item` instead.

[**Browser Support**](#)

While it could be used in one's implementation, this is not intended to serve as a polyfill for `Element#classList` or other instances of `DOMTokenList`.

The implementation of the `DOMTokenList` interface provided through `@wordpress/token-list` is broadly compatible in environments supporting ES5 (IE8 and newer). That being said, due to its internal implementation leveraging arrays for `TokenList#entries`, `TokenList#forEach`, `TokenList#keys`, and `TokenList#values`, you may need to assure that these functions are supported or polyfilled if you intend to use them.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/entries#Browser_compatibility
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/values#Browser_compatibility

`TokenList`'s own internal implementation of the `DOMTokenList` interface does not leverage any of these functions, so it is not necessary to polyfill them for basic usage.

[**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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/token-list](#)

[Previous: @wordpress/sync](#) [Previous: @wordpress/sync](#)

[Next: @wordpress/undo-manager](#) [Next: @wordpress/undo-manager](#)

@wordpress/undo-manager

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [createUndoManager](#)
- [Contributing to this package](#)

[↑ Back to top](#)

A simple undo manager.

[Installation](#)

Install the module

```
npm install @wordpress/undo-manager --save
```

[API](#)

[createUndoManager](#)

Creates an undo manager.

Returns

- `UndoManager`: Undo manager.

[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](#).

First published

September 11, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/undo-manager](#)

[Previous @wordpress/token-list](#) [Previous: @wordpress/token-list](#)

[Next @wordpress/url](#) [Next: @wordpress/url](#)

@wordpress/url

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
 - [addQueryArgs](#)
 - [buildQueryString](#)
 - [cleanForSlug](#)
 - [filterURLForDisplay](#)
 - [getAuthority](#)
 - [getFilename](#)
 - [getFragment](#)
 - [getPath](#)
 - [getPathAndQueryString](#)
 - [getProtocol](#)
 - [getQueryArg](#)
 - [getQueryArgs](#)
 - [getQueryString](#)
 - [hasQueryArg](#)
 - [isEmail](#)
 - [isURL](#)
 - [isValidAuthority](#)
 - [isValidFragment](#)
 - [isValidPath](#)
 - [isValidProtocol](#)
 - [isValidQueryString](#)
 - [normalizePath](#)
 - [prependHTTP](#)
 - [prependHTTPS](#)
 - [removeQueryArgs](#)
 - [safeDecodeURI](#)
 - [safeDecodeURIComponent](#)
- [Contributing to this package](#)

[↑ Back to top](#)

A collection of utilities to manipulate URLs.

[Installation](#)

Install the module

```
npm install @wordpress/url --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.

Usage

addQueryArgs

Appends arguments as querystring to the provided URL. If the URL already includes query arguments, the arguments are merged with (and take precedent over) the existing set.

Usage

```
const newURL = addQueryArgs( 'https://google.com', { q: 'test' } ); // https://google.com/?q=test
```

Parameters

- *url* [string]: URL to which arguments should be appended. If omitted, only the resulting querystring is returned.
- *args* [Object]: Query arguments to apply to URL.

Returns

- string: URL with arguments applied.

buildQueryString

Generates URL-encoded query string using input query data.

It is intended to behave equivalent as PHP's `http_build_query`, configured with encoding type `PHP_QUERY_RFC3986` (spaces as `%20`).

Usage

```
const queryString = buildQueryString( {
    simple: 'is ok',
    arrays: [ 'are', 'fine', 'too' ],
    objects: {
        evenNested: {
            ok: 'yes',
        },
    },
} );
// "simple=is%20ok&arrays%5B0%5D=are&arrays%5B1%5D=fine&arrays%5B2%5D=too&
```

Parameters

- *data* Record<string, *>: Data to encode.

Returns

- string: Query string.

[cleanForSlug](#)

Performs some basic cleanup of a string for use as a post slug.

This replicates some of what `sanitize_title()` does in WordPress core, but is only designed to approximate what the slug will be.

Converts Latin-1 Supplement and Latin Extended-A letters to basic Latin letters. Removes combining diacritical marks. Converts whitespace, periods, and forward slashes to hyphens. Removes any remaining non-word characters except hyphens. Converts remaining string to lowercase. It does not account for octets, HTML entities, or other encoded characters.

Parameters

- `string string`: Title or slug to be processed.

Returns

- `string`: Processed string.

[filterURLForDisplay](#)

Returns a URL for display.

Usage

```
const displayUrl = filterURLForDisplay(
  'https://www.wordpress.org/gutenberg/'
); // wordpress.org/gutenberg
const imageUrl = filterURLForDisplay(
  'https://www.wordpress.org/wp-content/uploads/img.png',
  20
); // ...ent/uploads/img.png
```

Parameters

- `url string`: Original URL.
- `maxLength number | null`: URL length.

Returns

- `string`: Displayed URL.

[getAuthority](#)

Returns the authority part of the URL.

Usage

```
const authority1 = getAuthority( 'https://wordpress.org/help/' ); // 'word
const authority2 = getAuthority( 'https://localhost:8080/test/' ); // 'loc
```

Parameters

- `url string`: The full URL.

Returns

- `string|void`: The authority part of the URL.

[getFilename](#)

Returns the filename part of the URL.

Usage

```
const filename1 = getFilename( 'http://localhost:8080/this/is/a/test.jpg' )
const filename2 = getFilename( '/this/is/a/test.png' ); // 'test.png'
```

Parameters

- `url string`: The full URL.

Returns

- `string|void`: The filename part of the URL.

[getFragment](#)

Returns the fragment part of the URL.

Usage

```
const fragment1 = getFragment(
    'http://localhost:8080/this/is/a/test?query=true#fragment'
); // '#fragment'
const fragment2 = getFragment(
    'https://wordpress.org#another-fragment?query=true'
); // '#another-fragment'
```

Parameters

- `url string`: The full URL

Returns

- `string|void`: The fragment part of the URL.

[getPath](#)

Returns the path part of the URL.

Usage

```
const path1 = getPath( 'http://localhost:8080/this/is/a/test?query=true' )
const path2 = getPath( 'https://wordpress.org/help/faq/' ); // 'help/faq'
```

Parameters

- `url string`: The full URL.

Returns

- `string|void`: The path part of the URL.

[getPathAndQueryString](#)

Returns the path part and query string part of the URL.

Usage

```
const pathAndQueryString1 = getPathAndQueryString(  
    'http://localhost:8080/this/is/a/test?query=true'  
); // '/this/is/a/test?query=true'  
const pathAndQueryString2 = getPathAndQueryString(  
    'https://wordpress.org/help/faq/'  
); // '/help/faq'
```

Parameters

- `url string`: The full URL.

Returns

- `string`: The path part and query string part of the URL.

[getProtocol](#)

Returns the protocol part of the URL.

Usage

```
const protocol1 = getProtocol( 'tel:012345678' ); // 'tel:'  
const protocol2 = getProtocol( 'https://wordpress.org' ); // 'https:'
```

Parameters

- `url string`: The full URL.

Returns

- `string|void`: The protocol part of the URL.

[getQueryArg](#)

Returns a single query argument of the url

Usage

```
const foo = getQueryArg( 'https://wordpress.org?foo=bar&bar=baz' , 'foo' );
```

Parameters

- `url string`: URL.
- `arg string`: Query arg name.

Returns

- `QueryArgParsed|void`: Query arg value.

[getQueryArgs](#)

Returns an object of query arguments of the given URL. If the given URL is invalid or has no querystring, an empty object is returned.

Usage

```
const foo = getQueryArgs( 'https://wordpress.org?foo=bar&bar=baz' );
// { "foo": "bar", "bar": "baz" }
```

Parameters

- `url string`: URL.

Returns

- `QueryArgs`: Query args object.

[getQueryString](#)

Returns the query string part of the URL.

Usage

```
const queryString = getQueryString(
  'http://localhost:8080/this/is/a/test?query=true#fragment'
); // 'query=true'
```

Parameters

- `url string`: The full URL.

Returns

- `string|void`: The query string part of the URL.

[hasQueryArg](#)

Determines whether the URL contains a given query arg.

Usage

```
const hasBar = hasQueryArg( 'https://wordpress.org?foo=bar&bar=baz' , 'bar' )
```

Parameters

- `url string`: URL.
- `arg string`: Query arg name.

Returns

- **boolean**: Whether or not the URL contains the query arg.

isEmail

Determines whether the given string looks like an email.

Usage

```
const isEmail = isEmail( 'hello@wordpress.org' ); // true
```

Parameters

- *email string*: The string to scrutinise.

Returns

- **boolean**: Whether or not it looks like an email.

isURL

Determines whether the given string looks like a URL.

Related

- <https://url.spec.whatwg.org/>
- <https://url.spec.whatwg.org/#valid-url-string>

Usage

```
const isURL = isURL( 'https://wordpress.org' ); // true
```

Parameters

- *url string*: The string to scrutinise.

Returns

- **boolean**: Whether or not it looks like a URL.

isValidAuthority

Checks for invalid characters within the provided authority.

Usage

```
const isValid = isValidAuthority( 'wordpress.org' ); // true
const isNotValid = isValidAuthority( 'wordpress#org' ); // false
```

Parameters

- *authority string*: A string containing the URL authority.

Returns

- **boolean**: True if the argument contains a valid authority.

isValidFragment

Checks for invalid characters within the provided fragment.

Usage

```
const isValid = isValidFragment( '#valid-fragment' ); // true
const isNotValid = isValidFragment( '#invalid-#fragment' ); // false
```

Parameters

- *fragment string*: The url fragment.

Returns

- **boolean**: True if the argument contains a valid fragment.

isValidPath

Checks for invalid characters within the provided path.

Usage

```
const isValid = isValidPath( 'test/path/' ); // true
const isNotValid = isValidPath( '/invalid?test/path/' ); // false
```

Parameters

- *path string*: The URL path.

Returns

- **boolean**: True if the argument contains a valid path

isValidProtocol

Tests if a url protocol is valid.

Usage

```
const isValid = isValidProtocol( 'https:' ); // true
const isNotValid = isValidProtocol( 'https ::' ); // false
```

Parameters

- *protocol string*: The url protocol.

Returns

- **boolean**: True if the argument is a valid protocol (e.g. http:, tel:).

[isValidQueryString](#)

Checks for invalid characters within the provided query string.

Usage

```
const isValid = isValidQueryString( 'query=true&another=false' ); // true
const isNotValid = isValidQueryString( 'query=true?another=false' ); // false
```

Parameters

- *queryString* **s t r i n g**: The query string.

Returns

- **boolean**: True if the argument contains a valid query string.

[normalizePath](#)

Given a path, returns a normalized path where equal query parameter values will be treated as identical, regardless of order they appear in the original text.

Parameters

- *path* **s t r i n g**: Original path.

Returns

- **s t r i n g**: Normalized path.

[prependHTTP](#)

Prepends “http://” to a url, if it looks like something that is meant to be a TLD.

Usage

```
const actualURL = prependHTTP( 'wordpress.org' ); // http://wordpress.org
```

Parameters

- *url* **s t r i n g**: The URL to test.

Returns

- **s t r i n g**: The updated URL.

[prependHTTPS](#)

Prepends “https://” to a url, if it looks like something that is meant to be a TLD.

Note: this will not replace “http://” with “<https://>”.

Usage

```
const actualURL = prependHTTPS( 'wordpress.org' ); // https://wordpress.or
```

Parameters

- *url* `string`: The URL to test.

Returns

- `string`: The updated URL.

[removeQueryArgs](#)

Removes arguments from the query string of the url

Usage

```
const newUrl = removeQueryArgs(  
  'https://wordpress.org?foo=bar&bar=baz&baz=foobar',  
  'foo',  
  'bar'  
) // https://wordpress.org?baz=foobar
```

Parameters

- *url* `string`: URL.
- *args* ... `string`: Query Args.

Returns

- `string`: Updated URL.

[safeDecodeURI](#)

Safely decodes a URI with `decodeURI`. Returns the URI unmodified if `decodeURI` throws an error.

Usage

```
const badUri = safeDecodeURI( '%z' ); // does not throw an Error, simply returns the original string
```

Parameters

- *uri* `string`: URI to decode.

Returns

- `string`: Decoded URI if possible.

[safeDecodeURIComponent](#)

Safely decodes a URI component with `decodeURIComponent`. Returns the URI component unmodified if `decodeURIComponent` throws an error.

Parameters

- *uriComponent* `string`: URI component to decode.

Returns

- `string`: Decoded URI component if possible.

[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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/url](#)

[Previous @wordpress/undo-manager](#) [Previous: @wordpress/undo-manager](#)
[Next @wordpress/viewport](#) [Next: @wordpress/viewport](#)

@wordpress/viewport

In this article

[Table of Contents](#)

- [Installation](#)
- [Usage](#)
 - [Data Module](#)
 - [Higher-Order Components](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Viewport is a module for responding to changes in the browser viewport size. It registers its own [data module](#), updated in response to browser media queries on a standard set of supported breakpoints. This data and the included higher-order components can be used in your own modules and components to implement viewport-dependent behaviors.

Installation

Install the module

```
npm install @wordpress/viewport --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.*

Usage

The standard set of breakpoint thresholds is as follows:

Name	Pixel Width
huge	1440
wide	1280
large	960
medium	782
small	600
mobile	480

Data Module

The Viewport module registers itself under the `core/viewport` data namespace and is exposed from the package as `store`.

```
import { select } from '@wordpress/data';
import { store } from '@wordpress/viewport';

const isSmall = select( store ).isViewportMatch( '< medium' );
```

The `isViewportMatch` selector accepts a single string argument `query`. It consists of an optional operator and breakpoint name, separated with a space. The operator can be `<` or `>=`, defaulting to `>=`.

```
import { select } from '@wordpress/data';
import { store } from '@wordpress/viewport';

const { isViewportMatch } = select( store );
const isSmall = isViewportMatch( '< medium' );
const isWideOrHuge = isViewportMatch( '>= wide' );
// Equivalent:
// const isWideOrHuge = isViewportMatch( 'wide' );
```

Higher-Order Components

This package provides a set of HOCs to author components whose behavior should vary depending on the viewport.

ifViewportMatches

Higher-order component creator, creating a new component which renders if the viewport query is satisfied.

Related

- `withViewportMatches`

Usage

```
function MyMobileComponent() {  
    return <div>I'm only rendered on mobile viewports!</div>;  
}  
  
MyMobileComponent = ifViewportMatches( '< small' )( MyMobileComponent );
```

Parameters

- *query string*: Viewport query.

Returns

- **Function**: Higher-order component.

store

Store definition for the viewport namespace.

Related

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

Type

- `Object`

withViewportMatch

Higher-order component creator, creating a new component which renders with the given prop names, where the value passed to the underlying component is the result of the query assigned as the object's value.

Related

- `isViewportMatch`

Usage

```
function MyComponent( { isMobile } ) {  
    return <div>Currently: { isMobile ? 'Mobile' : 'Not Mobile' }</div>;  
}  
  
MyComponent = withViewportMatch( { isMobile: '< small' } )( MyComponent );
```

Parameters

- `queries` Object: Object of prop name to viewport query.

Returns

- Function: Higher-order 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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/viewport](#)

[Previous](#) [@wordpress/url](#) [Previous: @wordpress/url](#)
[Next](#) [@wordpress/warning](#) [Next: @wordpress/warning](#)

@wordpress/warning

In this article

Table of Contents

- [Installation](#)
- [Reducing bundle size](#)
- [API](#)
 - [default](#)
- [Contributing to this package](#)

[↑ Back to top](#)

Utility for warning messages to the console based on a passed condition.

Installation

Install the module

```
npm install @wordpress/warning --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.*

Reducing bundle size

Literal strings aren't minified. Keeping them in your production bundle may increase the bundle size significantly.

To prevent that, you should:

1. Put `@wordpress/warning/babel-plugin` into your [babel config](#) or use [@wordpress/babel-preset-default](#), which already includes the babel plugin.

This will make sure your `warning` calls are wrapped within a condition that checks if `SCRIPT_DEBUG === true`.

2. Use [UglifyJS](#), [Terser](#) or any other JavaScript parser that performs [dead code elimination](#). This is usually used in conjunction with JavaScript bundlers, such as [webpack](#).

When parsing the code in production mode, the `warning` call will be removed altogether.

API

default

Shows a warning with `message` if environment is not `production`.

Usage

```
import warning from '@wordpress/warning';

function MyComponent( props ) {
    if ( ! props.title ) {
        warning( '`props.title` was not passed' );
    }
    ...
}
```

Parameters

- `message string`: Message to show in the warning.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/widgets](#)

[Previous @wordpress/widgets](#) [Previous: @wordpress/widgets](#)
[Next @wordpress/widgets](#) [Next: @wordpress/widgets](#)

@wordpress/widgets

In this article

[Table of Contents](#)

- [Installation](#)
- [Contributing to this package](#)

[↑ Back to top](#)

This package contains common functionality used by the widgets block editor in the Widgets screen and the Customizer.

This package is meant to be used only with WordPress core. Feel free to use it in your own project but please keep in mind that it might never get fully documented.

Installation

Install the module

```
npm install @wordpress/widgets
```

*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.*

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](#).

First published

May 4, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/widgets”](#)

[Previous @wordpress/warning](#) [Previous: @wordpress/warning](#)
[Next @wordpress/wordcount](#) [Next: @wordpress/wordcount](#)

@wordpress/wordcount

In this article

Table of Contents

- [Installation](#)
- [API](#)
 - [count](#)
- [Contributing to this package](#)

[↑ Back to top](#)

WordPress word count utility.

Installation

Install the module

```
npm install @wordpress/wordcount --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

count

Count some words.

Usage

```
import { count } from '@wordpress/wordcount';
const numberOfWorks = count( 'Words to count', 'words', {} );
```

Parameters

- *text* `string`: The text being processed
- *type* `WPWordCountStrategy`: The type of count. Accepts ‘words’, ‘characters_excluding_spaces’, or ‘characters_including_spaces’.
- *userSettings* `WPWordCountUserSettings`: Custom settings object.

Returns

- *number*: The word or character count.

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](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: @wordpress/wordcount”](#)

[Previous](#) [@wordpress/widgets](#) [Previous](#): [@wordpress/widgets](#)
[Next](#) [Data Module Reference](#) [Next](#): [Data Module Reference](#)

Data Module Reference

[↑ Back to top](#)

- [core: WordPress Core Data](#)
- [core/annotations: Annotations](#)
- [core/block-directory: Block directory](#)
- [core/block-editor: The Block Editor's Data](#)
- [core/blocks: Block Types Data](#)
- [core/commands: Command Palette](#)
- [core/customize-widgets: Customize Widgets](#)
- [core/edit-post: The Editor's UI Data](#)
- [core/edit-site: Edit Site](#)
- [core/edit-widgets: Edit Widgets](#)
- [core/editor: The Post Editor's Data](#)
- [core/keyboard-shortcuts: The Keyboard Shortcuts Data](#)
- [core/notices: Notices Data](#)
- [core/nux: The NUX \(New User Experience\) Data](#)
- [core/preferences: Preferences](#)
- [core/reusable-blocks: Reusable blocks](#)
- [core/rich-text: Rich Text](#)
- [core/viewport: The Viewport Data](#)

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Data Module Reference”](#)

[Previous](#) [@wordpress/wordcount](#) [Previous: @wordpress/wordcount](#)
[Next](#) [WordPress Core Data](#) [Next: WordPress Core Data](#)

WordPress Core Data

In this article

[Table of Contents](#)

- [Selectors
 - \[canUser\]\(#\)
 - \[canUserEditEntityRecord\]\(#\)
 - \[getAuthors\]\(#\)
 - \[getAutosave\]\(#\)](#)

- [getAutosaves](#)
- [getBlockPatternCategories](#)
- [getBlockPatterns](#)
- [getCurrentTheme](#)
- [getCurrentThemeGlobalStylesRevisions](#)
- [getCurrentUser](#)
- [getDefaultTemplateId](#)
- [getEditedEntityRecord](#)
- [getEmbedPreview](#)
- [getEntitiesByKind](#)
- [getEntitiesConfig](#)
- [getEntity](#)
- [getEntityConfig](#)
- [getEntityRecord](#)
- [getEntityRecordEdits](#)
- [getEntityRecordNonTransientEdits](#)
- [getEntityRecords](#)
- [getEntityRecordsTotalItems](#)
- [getEntityRecordsTotalPages](#)
- [getLastEntityDeleteError](#)
- [getLastEntitySaveError](#)
- [getRawEntityRecord](#)
- [getRedoEdit](#)
- [getReferenceByDistinctEdits](#)
- [getRevision](#)
- [getRevisions](#)
- [getThemeSupports](#)
- [getUndoEdit](#)
- [getUserPatternCategories](#)
- [getUserQueryResults](#)
- [hasEditsForEntityRecord](#)
- [hasEntityRecords](#)
- [hasFetchedAutosaves](#)
- [hasRedo](#)
- [hasUndo](#)
- [isAutosavingEntityRecord](#)
- [isDeletingEntityRecord](#)
- [isPreviewEmbedFallback](#)
- [isRequestingEmbedPreview](#)
- [isSavingEntityRecord](#)

- **[Actions](#)**

- [addEntities](#)
- [deleteEntityRecord](#)
- [editEntityRecord](#)
- [receiveDefaultTemplateId](#)
- [receiveEntityRecords](#)
- [receiveNavigationFallbackId](#)
- [receiveRevisions](#)
- [receiveThemeSupports](#)
- [receiveUploadPermissions](#)
- [redo](#)
- [saveEditedEntityRecord](#)
- [saveEntityRecord](#)
- [undo](#)

[↑ Back to top](#)

Namespace: `core`.

Selectors

canUser

Returns whether the current user can perform the given action on the given REST resource.

Calling this may trigger an OPTIONS request to the REST API via the `canUser()` resolver.

<https://developer.wordpress.org/rest-api/reference/>

Parameters

- `state` `State`: Data state.
- `action` `string`: Action to check. One of: ‘create’, ‘read’, ‘update’, ‘delete’.
- `resource` `string`: REST resource to check, e.g. ‘media’ or ‘posts’.
- `id` `EntityRecordKey`: Optional ID of the rest resource to check.

Returns

- `boolean` | `undefined`: Whether or not the user can perform the action, or `undefined` if the OPTIONS request is still being made.

canUserEditEntityRecord

Returns whether the current user can edit the given entity.

Calling this may trigger an OPTIONS request to the REST API via the `canUser()` resolver.

<https://developer.wordpress.org/rest-api/reference/>

Parameters

- `state` `State`: Data state.
- `kind` `string`: Entity kind.
- `name` `string`: Entity name.
- `recordId` `EntityRecordKey`: Record’s id.

Returns

- `boolean` | `undefined`: Whether or not the user can edit, or `undefined` if the OPTIONS request is still being made.

getAuthors

Deprecated since 11.3. Callers should use

`select('core').getUsers({ who: 'authors' })` instead.

Returns all available authors.

Parameters

- *state State*: Data state.
- *query GetRecordsHttpQuery*: Optional object of query parameters to include with request. For valid query parameters see the [Users page](#) in the REST API Handbook and see the arguments for [List Users](#) and [Retrieve a User](#).

Returns

- `ET.User[]`: Authors list.

[getAutosave](#)

Returns the autosave for the post and author.

Parameters

- *state State*: State tree.
- *postType string*: The type of the parent post.
- *postId EntityRecordKey*: The id of the parent post.
- *authorId EntityRecordKey*: The id of the author.

Returns

- `EntityRecord | undefined`: The autosave for the post and author.

[getAutosaves](#)

Returns the latest autosaves for the post.

May return multiple autosaves since the backend stores one autosave per author for each post.

Parameters

- *state State*: State tree.
- *postType string*: The type of the parent post.
- *postId EntityRecordKey*: The id of the parent post.

Returns

- `Array< any > | undefined`: An array of autosaves for the post, or undefined if there is none.

[getBlockPatternCategories](#)

Retrieve the list of registered block pattern categories.

Parameters

- *state State*: Data state.

Returns

- `Array< any >`: Block pattern category list.

[getBlockPatterns](#)

Retrieve the list of registered block patterns.

Parameters

- `state State`: Data state.

Returns

- `Array< any >`: Block pattern list.

[getCurrentTheme](#)

Return the current theme.

Parameters

- `state State`: Data state.

Returns

- `any`: The current theme.

[getCurrentThemeGlobalStylesRevisions](#)

Deprecated since WordPress 6.5.0. Callers should use

`select('core').getRevisions('root', 'globalStyles', ${ recordKey })` instead, where `recordKey` is the id of the global styles parent post.

Returns the revisions of the current global styles theme.

Parameters

- `state State`: Data state.

Returns

- `Array< object > | null`: The current global styles.

[getCurrentUser](#)

Returns the current user.

Parameters

- `state State`: Data state.

Returns

- `undefined< 'edit' >`: Current user object.

[getDefaultTemplateId](#)

Returns the default template use to render a given query.

Parameters

- *state State*: Data state.
- *query TemplateQuery*: Query.

Returns

- *string*: The default template id for the given query.

[getEditedEntityRecord](#)

Returns the specified entity record, merged with its edits.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- *undefined< EntityRecord > | undefined*: The entity record, merged with its edits.

[getEmbedPreview](#)

Returns the embed preview for the given URL.

Parameters

- *state State*: Data state.
- *url string*: Embedded URL.

Returns

- *any*: Undefined if the preview has not been fetched, otherwise, the preview fetched from the embed preview API.

[getEntitiesByKind](#)

Deprecated since WordPress 6.0. Use `getEntitiesConfig` instead

Returns the loaded entities for the given kind.

Parameters

- *state State*: Data state.
- *kind string*: Entity kind.

Returns

- `Array< any >`: Array of entities with config matching kind.

[getEntitiesConfig](#)

Returns the loaded entities for the given kind.

Parameters

- `state State`: Data state.
- `kind string`: Entity kind.

Returns

- `Array< any >`: Array of entities with config matching kind.

[getEntity](#)

Deprecated since WordPress 6.0. Use `getEntityConfig` instead

Returns the entity config given its kind and name.

Parameters

- `state State`: Data state.
- `kind string`: Entity kind.
- `name string`: Entity name.

Returns

- `any`: Entity config

[getEntityConfig](#)

Returns the entity config given its kind and name.

Parameters

- `state State`: Data state.
- `kind string`: Entity kind.
- `name string`: Entity name.

Returns

- `any`: Entity config

[getEntityRecord](#)

Returns the Entity's record object by key. Returns `null` if the value is not yet received, `undefined` if the value entity is known to not exist, or the entity object if it exists and is received.

Parameters

- `state State`: State tree

- *kind* **string**: Entity kind.
- *name* **string**: Entity name.
- *key* **EntityRecordKey**: Record's key
- *query* **GetRecordsHttpQuery**: Optional query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available "Retrieve a [Entity kind]".

Returns

- **EntityRecord** | **undefined**: Record.

[getEntityRecordEdits](#)

Returns the specified entity record's edits.

Parameters

- *state* **State**: State tree.
- *kind* **string**: Entity kind.
- *name* **string**: Entity name.
- *recordId* **EntityRecordKey**: Record ID.

Returns

- **Optional< any >**: The entity record's edits.

[getEntityRecordNonTransientEdits](#)

Returns the specified entity record's non transient edits.

Transient edits don't create an undo level, and are not considered for change detection. They are defined in the entity's config.

Parameters

- *state* **State**: State tree.
- *kind* **string**: Entity kind.
- *name* **string**: Entity name.
- *recordId* **EntityRecordKey**: Record ID.

Returns

- **Optional< any >**: The entity record's non transient edits.

[getEntityRecords](#)

Returns the Entity's records.

Parameters

- *state* **State**: State tree
- *kind* **string**: Entity kind.
- *name* **string**: Entity name.

- *query GetRecordsHttpQuery*: Optional terms query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for “List [Entity kind]s”.

Returns

- EntityRecord[] | null: Records.

[**getEntityRecordsTotalItems**](#)

Returns the Entity’s total available records for a given query (ignoring pagination).

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *query GetRecordsHttpQuery*: Optional terms query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for “List [Entity kind]s”.

Returns

- number | null: number | null.

[**getEntityRecordsTotalPages**](#)

Returns the number of available pages for the given query.

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *query GetRecordsHttpQuery*: Optional terms query. If requesting specific fields, fields must always include the ID. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for “List [Entity kind]s”.

Returns

- number | null: number | null.

[**getLastEntityDeleteError**](#)

Returns the specified entity record’s last delete error.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.

- *recordId EntityRecordKey*: Record ID.

Returns

- any: The entity record's save error.

[getLastError](#)

Returns the specified entity record's last save error.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- any: The entity record's save error.

[getRawEntityRecord](#)

Returns the entity's record object by key, with its attributes mapped to their raw values.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *key EntityRecordKey*: Record's key.

Returns

- EntityRecord | undefined: Object with the entity's raw attributes.

[getRedoEdit](#)

Deprecated since 6.3

Returns the next edit from the current undo offset for the entity records edits history, if any.

Parameters

- *state State*: State tree.

Returns

- Optional< any >: The edit.

[getReferenceByDistinctEdits](#)

Returns a new reference when edited values have changed. This is useful in inferring where an edit has been made between states by comparison of the return values using strict equality.

Usage

```
const hasEditOccurred = (
  getReferenceByDistinctEdits( beforeState ) !==
  getReferenceByDistinctEdits( afterState )
);
```

Parameters

- *state* Editor state.

Returns

- A value whose reference will change only when an edit occurs.

getRevision

Returns a single, specific revision of a parent entity.

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordKey EntityRecordKey*: The key of the entity record whose revisions you want to fetch.
- *revisionKey EntityRecordKey*: The revision's key.
- *query GetRecordsHttpQuery*: Optional query. If requesting specific fields, fields must always include the ID. For valid query parameters see revisions schema in [the REST API Handbook](#). Then see the arguments available “Retrieve a [entity kind]”.

Returns

- `RevisionRecord | Record< PropertyKey, never > | undefined: Record`.

getRevisions

Returns an entity's revisions.

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordKey EntityRecordKey*: The key of the entity record whose revisions you want to fetch.
- *query GetRecordsHttpQuery*: Optional query. If requesting specific fields, fields must always include the ID. For valid query parameters see revisions schema in [the REST API Handbook](#). Then see the arguments available “Retrieve a [Entity kind]”.

Returns

- `RevisionRecord[] | null: Record`.

[getThemeSupports](#)

Return theme supports data in the index.

Parameters

- *state State*: Data state.

Returns

- *any*: Index data.

[getUndoEdit](#)

Deprecated since 6.3

Returns the previous edit from the current undo offset for the entity records edits history, if any.

Parameters

- *state State*: State tree.

Returns

- *Optional< any >*: The edit.

[getUserPatternCategories](#)

Retrieve the registered user pattern categories.

Parameters

- *state State*: Data state.

Returns

- *Array< UserPatternCategory >*: User patterns category array.

[getUserQueryResults](#)

Returns all the users returned by a query ID.

Parameters

- *state State*: Data state.
- *queryID string*: Query ID.

Returns

- *undefined< 'edit' >[]*: Users list.

[hasEditsForEntityRecord](#)

Returns true if the specified entity record has edits, and false otherwise.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- boolean: Whether the entity record has edits or not.

[hasEntityRecords](#)

Returns true if records have been received for the given set of parameters, or false otherwise.

Parameters

- *state State*: State tree
- *kind string*: Entity kind.
- *name string*: Entity name.
- *query GetRecordsHttpQuery*: Optional terms query. For valid query parameters see the [Reference](#) in the REST API Handbook and select the entity kind. Then see the arguments available for “List [Entity kind]s”.

Returns

- boolean: Whether entity records have been received.

[hasFetchedAutosaves](#)

Returns true if the REST request for autosaves has completed.

Parameters

- *state State*: State tree.
- *postType string*: The type of the parent post.
- *postId EntityRecordKey*: The id of the parent post.

Returns

- boolean: True if the REST request was completed. False otherwise.

[hasRedo](#)

Returns true if there is a next edit from the current undo offset for the entity records edits history, and false otherwise.

Parameters

- *state State*: State tree.

Returns

- boolean: Whether there is a next edit or not.

[hasUndo](#)

Returns true if there is a previous edit from the current undo offset for the entity records edits history, and false otherwise.

Parameters

- *state State*: State tree.

Returns

- boolean: Whether there is a previous edit or not.

[isAutosavingEntityRecord](#)

Returns true if the specified entity record is autosaving, and false otherwise.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- boolean: Whether the entity record is autosaving or not.

[isDeletingEntityRecord](#)

Returns true if the specified entity record is deleting, and false otherwise.

Parameters

- *state State*: State tree.
- *kind string*: Entity kind.
- *name string*: Entity name.
- *recordId EntityRecordKey*: Record ID.

Returns

- boolean: Whether the entity record is deleting or not.

[isPreviewEmbedFallback](#)

Determines if the returned preview is an oEmbed link fallback.

WordPress can be configured to return a simple link to a URL if it is not embeddable. We need to be able to determine if a URL is embeddable or not, based on what we get back from the oEmbed preview API.

Parameters

- *state State*: Data state.
- *url string*: Embedded URL.

Returns

- `boolean`: Is the preview for the URL an oEmbed link fallback.

[isRequestingEmbedPreview](#)

Returns true if a request is in progress for embed preview data, or false otherwise.

Parameters

- `state State`: Data state.
- `url string`: URL the preview would be for.

Returns

- `boolean`: Whether a request is in progress for an embed preview.

[isSavingEntityRecord](#)

Returns true if the specified entity record is saving, and false otherwise.

Parameters

- `state State`: State tree.
- `kind string`: Entity kind.
- `name string`: Entity name.
- `recordId EntityRecordKey`: Record ID.

Returns

- `boolean`: Whether the entity record is saving or not.

[Actions](#)

[addEntities](#)

Returns an action object used in adding new entities.

Parameters

- `entities Array`: Entities received.

Returns

- `Object`: Action object.

[deleteEntityRecord](#)

Action triggered to delete an entity record.

Parameters

- `kind string`: Kind of the deleted entity.
- `name string`: Name of the deleted entity.

- *recordId* `string`: Record ID of the deleted entity.
- *query* `?Object`: Special query parameters for the DELETE API call.
- *options* `[Object]`: Delete options.
- *options._unstableFetch* `[Function]`: Internal use only. Function to call instead of `apiFetch()`. Must return a promise.
- *options.throwOnError* `[boolean]`: If false, this action suppresses all the exceptions. Defaults to false.

[editEntityRecord](#)

Returns an action object that triggers an edit to an entity record.

Parameters

- *kind* `string`: Kind of the edited entity record.
- *name* `string`: Name of the edited entity record.
- *recordId* `number|string`: Record ID of the edited entity record.
- *edits* `Object`: The edits.
- *options* `Object`: Options for the edit.
- *options.undoIgnore* `[boolean]`: Whether to ignore the edit in undo history or not.

Returns

- `Object`: Action object.

[receiveDefaultTemplateId](#)

Returns an action object used to set the template for a given query.

Parameters

- *query* `Object`: The lookup query.
- *templateId* `string`: The resolved template id.

Returns

- `Object`: Action object.

[receiveEntityRecords](#)

Returns an action object used in signalling that entity records have been received.

Parameters

- *kind* `string`: Kind of the received entity record.
- *name* `string`: Name of the received entity record.
- *records* `Array|Object`: Records received.
- *query* `?Object`: Query Object.
- *invalidateCache* `?boolean`: Should invalidate query caches.
- *edits* `?Object`: Edits to reset.
- *meta* `?Object`: Meta information about pagination.

Returns

- Object: Action object.

[receiveNavigationFallbackId](#)

Returns an action object signalling that the fallback Navigation Menu id has been received.

Parameters

- *fallbackId* integer: the id of the fallback Navigation Menu

Returns

- Object: Action object.

[receiveRevisions](#)

Action triggered to receive revision items.

Parameters

- *kind* string: Kind of the received entity record revisions.
- *name* string: Name of the received entity record revisions.
- *recordKey* number | string: The key of the entity record whose revisions you want to fetch.
- *records* Array | Object: Revisions received.
- *query* ?Object: Query Object.
- *invalidateCache* ?boolean: Should invalidate query caches.
- *meta* ?Object: Meta information about pagination.

[receiveThemeSupports](#)

Deprecated since WP 5.9, this is not useful anymore, use the selector directly.

Returns an action object used in signalling that the index has been received.

Returns

- Object: Action object.

[receiveUploadPermissions](#)

Deprecated since WP 5.9, use `receiveUserPermission` instead.

Returns an action object used in signalling that Upload permissions have been received.

Parameters

- *hasUploadPermissions* boolean: Does the user have permission to upload files?

Returns

- Object: Action object.

[redo](#)

Action triggered to redo the last undoed edit to an entity record, if any.

[saveEditedEntityRecord](#)

Action triggered to save an entity record's edits.

Parameters

- *kind* `string`: Kind of the entity.
- *name* `string`: Name of the entity.
- *recordId* `Object`: ID of the record.
- *options* `Object`: Saving options.

[saveEntityRecord](#)

Action triggered to save an entity record.

Parameters

- *kind* `string`: Kind of the received entity.
- *name* `string`: Name of the received entity.
- *record* `Object`: Record to be saved.
- *options* `Object`: Saving options.
- *options.isAutosave* `[boolean]`: Whether this is an autosave.
- *options._unstableFetch* `[Function]`: Internal use only. Function to call instead of `apiFetch()`. Must return a promise.
- *options.throwOnError* `[boolean]`: If false, this action suppresses all the exceptions. Defaults to false.

[undo](#)

Action triggered to undo the last edit to an entity record, if any.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: WordPress Core Data”](#)

[Previous Data Module Reference](#) [Previous: Data Module Reference](#)
[Next Annotations](#) [Next: Annotations](#)

Annotations

In this article

Table of Contents

- [Selectors](#)
- [Actions](#)

[↑ Back to top](#)

Namespace: `core/annotations`.

This package is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

[Selectors](#)

Nothing to document.

[Actions](#)

Nothing to document.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Annotations”](#)

[Previous WordPress Core Data](#) [Previous: WordPress Core Data](#)

[Next Block directory](#) [Next: Block directory](#)

Block directory

In this article

Table of Contents

- [Selectors](#)
 - [getDownloadableBlocks](#)
 - [getErrorNoticeForBlock](#)

- [getErrorNotices](#)
- [getInstalledBlockTypes](#)
- [getNewBlockTypes](#)
- [getUnusedBlockTypes](#)
- [isInstalling](#)
- [isRequestingDownloadableBlocks](#)
- [Actions](#)
 - [addInstalledBlockType](#)
 - [clearErrorNotice](#)
 - [fetchDownloadableBlocks](#)
 - [installBlockType](#)
 - [receiveDownloadableBlocks](#)
 - [removeInstalledBlockType](#)
 - [setErrorNotice](#)
 - [setIsInstalling](#)
 - [uninstallBlockType](#)

[↑ Back to top](#)

Namespace: core/block-directory.

Selectors

getDownloadableBlocks

Returns the available uninstalled blocks.

Parameters

- *state Object*: Global application state.
- *filterValue string*: Search string.

Returns

- *Array*: Downloadable blocks.

getErrorNoticeForBlock

Returns the error notice for a given block.

Parameters

- *state Object*: Global application state.
- *blockId string*: The ID of the block plugin. eg: my-block

Returns

- *string | boolean*: The error text, or false if no error.

getErrorNotices

Returns all block error notices.

Parameters

- *state Object*: Global application state.

Returns

- *Object*: Object with error notices.

[getInstalledBlockTypes](#)

Returns the block types that have been installed on the server in this session.

Parameters

- *state Object*: Global application state.

Returns

- *Array*: Block type items

[getNewBlockTypes](#)

Returns block types that have been installed on the server and used in the current post.

Parameters

- *state Object*: Global application state.

Returns

- *Array*: Block type items.

[getUnusedBlockTypes](#)

Returns the block types that have been installed on the server but are not used in the current post.

Parameters

- *state Object*: Global application state.

Returns

- *Array*: Block type items.

[isInstalling](#)

Returns true if a block plugin install is in progress.

Parameters

- *state Object*: Global application state.
- *blockId string*: Id of the block.

Returns

- `boolean`: Whether this block is currently being installed.

[isRequestingDownloadableBlocks](#)

Returns true if application is requesting for downloadable blocks.

Parameters

- `state Object`: Global application state.
- `filterValue string`: Search string.

Returns

- `boolean`: Whether a request is in progress for the blocks list.

[Actions](#)

[addInstalledBlockType](#)

Returns an action object used to add a block type to the “newly installed” tracking list.

Parameters

- `item Object`: The block item with the block id and name.

Returns

- `Object`: Action object.

[clearErrorNotice](#)

Sets the error notice to empty for specific block.

Parameters

- `blockId string`: The ID of the block plugin. eg: my-block

Returns

- `Object`: Action object.

[fetchDownloadableBlocks](#)

Returns an action object used in signalling that the downloadable blocks have been requested and are loading.

Parameters

- `filterValue string`: Search string.

Returns

- `Object`: Action object.

[installBlockType](#)

Action triggered to install a block plugin.

Parameters

- `block Object`: The block item returned by search.

Returns

- `boolean`: Whether the block was successfully installed & loaded.

[receiveDownloadableBlocks](#)

Returns an action object used in signalling that the downloadable blocks have been updated.

Parameters

- `downloadableBlocks Array`: Downloadable blocks.
- `filterValue string`: Search string.

Returns

- `Object`: Action object.

[removeInstalledBlockType](#)

Returns an action object used to remove a block type from the “newly installed” tracking list.

Parameters

- `item string`: The block item with the block id and name.

Returns

- `Object`: Action object.

[setErrorNotice](#)

Sets an error notice to be displayed to the user for a given block.

Parameters

- `blockId string`: The ID of the block plugin. eg: my-block
- `message string`: The message shown in the notice.
- `isFatal boolean`: Whether the user can recover from the error.

Returns

- `Object`: Action object.

[setIsInstalling](#)

Returns an action object used to indicate install in progress.

Parameters

- *blockId* string:
- *isInstalling* boolean:

Returns

- Object: Action object.

[uninstallBlockType](#)

Action triggered to uninstall a block plugin.

Parameters

- *blockObject*: The blockType object.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Block directory](#)”

[Previous Annotations](#) [Previous: Annotations](#)

[Next The Block Editor’s Data](#) [Next: The Block Editor’s Data](#)

The Block Editor’s Data

In this article

Table of Contents

- [Selectors](#)
 - [areInnerBlocksControlled](#)
 - [canEditBlock](#)
 - [canInsertBlocks](#)
 - [canInsertBlockType](#)
 - [canLockBlockType](#)
 - [canMoveBlock](#)
 - [canMoveBlocks](#)
 - [canRemoveBlock](#)
 - [canRemoveBlocks](#)
 - [didAutomaticChange](#)

- [getAdjacentBlockClientId](#)
- [getAllowedBlocks](#)
- [getBlock](#)
- [getBlockAttributes](#)
- [getBlockCount](#)
- [getBlockEditingMode](#)
- [getBlockHierarchyRootClientId](#)
- [getBlockIndex](#)
- [getBlockInsertionPoint](#)
- [getBlockListSettings](#)
- [getBlockMode](#)
- [getBlockName](#)
- [getBlockNamesByClientId](#)
- [getBlockOrder](#)
- [getBlockParents](#)
- [getBlockParentsByBlockName](#)
- [getBlockRootClientId](#)
- [getBlocks](#)
- [getBlocksByClientId](#)
- [getBlocksByName](#)
- [getBlockSelectionEnd](#)
- [getBlockSelectionStart](#)
- [getBlockTransformItems](#)
- [getClientIdsOfDescendants](#)
- [getClientIdsWithDescendants](#)
- [getDirectInsertBlock](#)
- [getDraggedBlockClientIds](#)
- [getFirstMultiSelectedBlockClientId](#)
- [getGlobalBlockCount](#)
- [getInserterItems](#)
- [getLastMultiSelectedBlockClientId](#)
- [getLowestCommonAncestorWithSelectedBlock](#)
- [getMultiSelectedBlockClientIds](#)
- [getMultiSelectedBlocks](#)
- [getMultiSelectedBlocksEndClientId](#)
- [getMultiSelectedBlocksStartClientId](#)
- [getNextBlockClientId](#)
- [getPatternsByBlockTypes](#)
- [getPreviousBlockClientId](#)
- [getSelectedBlock](#)
- [getSelectedBlockClientId](#)
- [getSelectedBlockClientIds](#)
- [getSelectedBlockCount](#)
- [getSelectedBlocksInitialCaretPosition](#)
- [getSelectionEnd](#)
- [getSelectionStart](#)
- [getSettings](#)
- [getTemplate](#)
- [getTemplateLock](#)
- [hasBlockMovingClientId](#)
- [hasDraggedInnerBlock](#)
- [hasInserterItems](#)
- [hasMultiSelection](#)
- [hasSelectedBlock](#)

- [hasSelectedInnerBlock](#)
- [isAncestorBeingDragged](#)
- [isAncestorMultiSelected](#)
- [isBlockBeingDragged](#)
- [isBlockHighlighted](#)
- [isBlockInsertionPointVisible](#)
- [isBlockMultiSelected](#)
- [isBlockSelected](#)
- [isBlockValid](#)
- [isBlockVisible](#)
- [isBlockWithinSelection](#)
- [isCaretWithinFormattedText](#)
- [isDraggingBlocks](#)
- [isFirstMultiSelectedBlock](#)
- [isGroupable](#)
- [isLastBlockChangePersistent](#)
- [isMultiSelecting](#)
- [isNavigationMode](#)
- [isSelectionEnabled](#)
- [isTyping](#)
- [isUngroupable](#)
- [isValidTemplate](#)
- [wasBlockJustInserted](#)

- [Actions](#)

- [clearSelectedBlock](#)
- [duplicateBlocks](#)
- [enterFormattedText](#)
- [exitFormattedText](#)
- [flashBlock](#)
- [hideInsertionPoint](#)
- [insertAfterBlock](#)
- [insertBeforeBlock](#)
- [insertBlock](#)
- [insertBlocks](#)
- [insertDefaultBlock](#)
- [mergeBlocks](#)
- [moveBlocksDown](#)
- [moveBlocksToPosition](#)
- [moveBlocksUp](#)
- [moveBlockToPosition](#)
- [multiSelect](#)
- [receiveBlocks](#)
- [registerInserterMediaCategory](#)
- [removeBlock](#)
- [removeBlocks](#)
- [replaceBlock](#)
- [replaceBlocks](#)
- [replaceInnerBlocks](#)
- [resetBlocks](#)
- [resetSelection](#)
- [selectBlock](#)
- [selectionChange](#)
- [selectNextBlock](#)
- [selectPreviousBlock](#)

- [setBlockEditMode](#)
- [setBlockMovingClientId](#)
- [setBlockVisibility](#)
- [setHasControlledInnerBlocks](#)
- [setNavigationMode](#)
- [setTemplateValidity](#)
- [showInsertionPoint](#)
- [startDraggingBlocks](#)
- [startMultiSelect](#)
- [startTyping](#)
- [stopDraggingBlocks](#)
- [stopMultiSelect](#)
- [stopTyping](#)
- [synchronizeTemplate](#)
- [toggleBlockHighlight](#)
- [toggleBlockMode](#)
- [toggleSelection](#)
- [unsetBlockEditMode](#)
- [updateBlock](#)
- [updateBlockAttributes](#)
- [updateBlockListSettings](#)
- [updateSettings](#)
- [validateBlocksToTemplate](#)

[↑ Back to top](#)

Namespace: core/block-editor.

Selectors

areInnerBlocksControlled

Checks if a given block has controlled inner blocks.

Parameters

- *state Object*: Global application state.
- *clientId string*: The block to check.

Returns

- *boolean*: True if the block has controlled inner blocks.

canEditBlock

Determines if the given block is allowed to be edited.

Parameters

- *state Object*: Editor state.
- *clientId string*: The block client Id.

Returns

- `boolean`: Whether the given block is allowed to be edited.

[canInsertBlocks](#)

Determines if the given blocks are allowed to be inserted into the block list.

Parameters

- `state Object`: Editor state.
- `clientIds string`: The block client IDs to be inserted.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `boolean`: Whether the given blocks are allowed to be inserted.

[canInsertBlockType](#)

Determines if the given block type is allowed to be inserted into the block list.

Parameters

- `state Object`: Editor state.
- `blockName string`: The name of the block type, e.g. 'core/paragraph'.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `boolean`: Whether the given block type is allowed to be inserted.

[canLockBlockType](#)

Determines if the given block type can be locked/unlocked by a user.

Parameters

- `state Object`: Editor state.
- `nameOrType (string|Object)`: Block name or type object.

Returns

- `boolean`: Whether a given block type can be locked/unlocked.

[canMoveBlock](#)

Determines if the given block is allowed to be moved.

Parameters

- `state Object`: Editor state.
- `clientId string`: The block client Id.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `boolean | undefined`: Whether the given block is allowed to be moved.

[canMoveBlocks](#)

Determines if the given blocks are allowed to be moved.

Parameters

- `state Object`: Editor state.
- `clientIds string`: The block client IDs to be moved.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `boolean`: Whether the given blocks are allowed to be moved.

[canRemoveBlock](#)

Determines if the given block is allowed to be deleted.

Parameters

- `state Object`: Editor state.
- `clientId string`: The block client Id.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `boolean`: Whether the given block is allowed to be removed.

[canRemoveBlocks](#)

Determines if the given blocks are allowed to be removed.

Parameters

- `state Object`: Editor state.
- `clientIds string`: The block client IDs to be removed.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `boolean`: Whether the given blocks are allowed to be removed.

[didAutomaticChange](#)

Returns true if the last change was an automatic change, false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether the last change was automatic.

[getAdjacentBlockClientId](#)

Returns the client ID of the block adjacent one at the given reference `startClientId` and modifier directionality. Defaults start `startClientId` to the selected block, and direction as next block. Returns null if there is no adjacent block.

Parameters

- `state Object`: Editor state.
- `startClientId ?string`: Optional client ID of block from which to search.
- `modifier ?number`: Directionality multiplier (1 next, -1 previous).

Returns

- `?string`: Return the client ID of the block, or null if none exists.

[getAllowedBlocks](#)

Returns the list of allowed inserter blocks for inner blocks children.

Parameters

- `state Object`: Editor state.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `Array ?: The list of allowed block types.`

[getBlock](#)

Returns a block given its client ID. This is a parsed copy of the block, containing its `blockName`, `clientId`, and current `attributes` state. This is not the block's registration settings, which must be retrieved from the blocks module registration store.

`getBlock` recurses through its inner blocks until all its children blocks have been retrieved. Note that `getBlock` will not return the child inner blocks of an inner block controller. This is because an inner block controller syncs itself with its own entity, and should therefore not be included with the blocks of a different entity. For example, say you call `getBlocks(TP)` to get the blocks of a template part. If another template part is a child of `TP`, then the nested template part's child blocks will not be returned. This way, the template block itself is considered part of the parent, but the children are not.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `Object`: Parsed block object.

[getBlockAttributes](#)

Returns a block's attributes given its client ID, or null if no block exists with the client ID.

Parameters

- *state Object*: Editor state.
- *clientId string*: Block client ID.

Returns

- *Object?*: Block attributes.

[getBlockCount](#)

Returns the number of blocks currently present in the post.

Parameters

- *state Object*: Editor state.
- *rootClientId ?string*: Optional root client ID of block list.

Returns

- *number*: Number of blocks in the post.

[getBlockEditingStyle](#)

Returns the block editing mode for a given block.

The mode can be one of three options:

- '*disabled*': Prevents editing the block entirely, i.e. it cannot be selected.
- '*contentOnly*': Hides all non-content UI, e.g. auxiliary controls in the toolbar, the block movers, block settings.
- '*default*': Allows editing the block as normal.

Blocks can set a mode using the `useBlockEditingStyle` hook.

The mode is inherited by all of the block's inner blocks, unless they have their own mode.

A template lock can also set a mode. If the template lock is '`contentOnly`', the block's mode is overridden to '`contentOnly`' if the block has a content role attribute, or '`disabled`' otherwise.

Related

- `useBlockEditingStyle`

Parameters

- *state Object*: Global application state.
- *clientId string*: The block client ID, or '' for the root container.

Returns

- `BlockEditingMode`: The block editing mode. One of 'disabled', 'contentOnly', or 'default'.

[**getBlockHierarchyRootClientId**](#)

Given a block client ID, returns the root of the hierarchy from which the block is nested, return the block itself for root level blocks.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block from which to find root client ID.

Returns

- `string`: Root client ID

[**getBlockIndex**](#)

Returns the index at which the block corresponding to the specified client ID occurs within the block order, or -1 if the block does not exist.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `number`: Index at which block exists in order.

[**getBlockInsertionPoint**](#)

Returns the insertion point, the index at which the new inserted block would be placed. Defaults to the last index.

Parameters

- `state Object`: Editor state.

Returns

- `Object`: Insertion point object with `rootClientId`, `index`.

[**getBlockListSettings**](#)

Returns the Block List settings of a block, if any exist.

Parameters

- `state Object`: Editor state.
- `clientId ?string`: Block client ID.

Returns

- `?Object`: Block settings of the block if set.

[getBlockMode](#)

Returns the block's editing mode, defaulting to "visual" if not explicitly assigned.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `Object`: Block editing mode.

[getBlockName](#)

Returns a block's name given its client ID, or null if no block exists with the client ID.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `string`: Block name.

[getBlockNamesByClientId](#)

Given an array of block client IDs, returns the corresponding array of block names.

Parameters

- `state Object`: Editor state.
- `clientIds string []`: Client IDs for which block names are to be returned.

Returns

- `string []`: Block names.

[getBlockOrder](#)

Returns an array containing all block client IDs in the editor in the order they appear. Optionally accepts a root client ID of the block list for which the order should be returned, defaulting to the top-level block order.

Parameters

- `state Object`: Editor state.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- **Array**: Ordered client IDs of editor blocks.

[getBlockParents](#)

Given a block client ID, returns the list of all its parents from top to bottom.

Parameters

- *state Object*: Editor state.
- *clientId string*: Block from which to find root client ID.
- *ascending boolean*: Order results from bottom to top (true) or top to bottom (false).

Returns

- **Array**: ClientIDs of the parent blocks.

[getBlockParentsByBlockName](#)

Given a block client ID and a block name, returns the list of all its parents from top to bottom, filtered by the given name(s). For example, if passed ‘core/group’ as the blockName, it will only return parents which are group blocks. If passed [‘core/group’, ‘core/cover’], as the blockName, it will return parents which are group blocks and parents which are cover blocks.

Parameters

- *state Object*: Editor state.
- *clientId string*: Block from which to find root client ID.
- *blockName string|string[]*: Block name(s) to filter.
- *ascending boolean*: Order results from bottom to top (true) or top to bottom (false).

Returns

- **Array**: ClientIDs of the parent blocks.

[getBlockRootClientId](#)

Given a block client ID, returns the root block from which the block is nested, an empty string for top-level blocks, or null if the block does not exist.

Parameters

- *state Object*: Editor state.
- *clientId string*: Block from which to find root client ID.

Returns

- **?string**: Root client ID, if exists

[getBlocks](#)

Returns all block objects for the current post being edited as an array in the order they appear in the post. Note that this will exclude child blocks of nested inner block controllers.

Parameters

- *state Object*: Editor state.
- *rootClientId ?string*: Optional root client ID of block list.

Returns

- *Object []*: Post blocks.

[getBlocksByClientId](#)

Given an array of block client IDs, returns the corresponding array of block objects.

Parameters

- *state Object*: Editor state.
- *clientIds string []*: Client IDs for which blocks are to be returned.

Returns

- *WPBlock []*: Block objects.

[getBlocksByName](#)

Returns all blocks that match a blockName. Results include nested blocks.

Parameters

- *state Object*: Global application state.
- *blockName ?string*: Optional block name, if not specified, returns an empty array.

Returns

- *Array*: Array of clientIds of blocks with name equal to blockName.

[getBlockSelectionEnd](#)

Returns the current block selection end. This value may be null, and it may represent either a singular block selection or multi-selection end. A selection is singular if its start and end match.

Parameters

- *state Object*: Global application state.

Returns

- *?string*: Client ID of block selection end.

[getBlockSelectionStart](#)

Returns the current block selection start. This value may be null, and it may represent either a singular block selection or multi-selection start. A selection is singular if its start and end match.

Parameters

- *state Object*: Global application state.

Returns

- ?string: Client ID of block selection start.

[getBlockTransformItems](#)

Determines the items that appear in the available block transforms list.

Each item object contains what's necessary to display a menu item in the transform list and handle its selection.

The ‘frecency’ property is a heuristic (<https://en.wikipedia.org/wiki/Frecency>) that combines block usage frequency and recency.

Items are returned ordered descendingly by their ‘frecency’.

Parameters

- *state Object*: Editor state.
- *blocks Object|Object[]*: Block object or array objects.
- *rootClientId ?string*: Optional root client ID of block list.

Returns

- *WPEditorTransformItem[]*: Items that appear in inserter.

Type Definition

- *WPEditorTransformItem Object*

Properties

- *id string*: Unique identifier for the item.
- *name string*: The type of block to create.
- *title string*: Title of the item, as it appears in the inserter.
- *icon string*: Dashicon for the item, as it appears in the inserter.
- *isDisabled boolean*: Whether or not the user should be prevented from inserting this item.
- *frecency number*: Heuristic that combines frequency and recency.

[getClientIdsOfDescendants](#)

Returns an array containing the clientIds of all descendants of the blocks given. Returned ids are ordered first by the order of the ids given, then by the order that they appear in the editor.

Parameters

- *state Object*: Global application state.
- *rootIds string|string[]*: Client ID(s) for which descendant blocks are to be returned.

Returns

- `Array`: Client IDs of descendants.

[getClientIdsWithDescendants](#)

Returns an array containing the clientIDs of the top-level blocks and their descendants of any depth (for nested blocks). Ids are returned in the same order that they appear in the editor.

Parameters

- `state Object`: Global application state.

Returns

- `Array`: ids of top-level and descendant blocks.

[getDirectInsertBlock](#)

Returns the block to be directly inserted by the block appender.

Parameters

- `state Object`: Editor state.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `?WPDirectInsertBlock`: The block type to be directly inserted.

Type Definition

- `WPDirectInsertBlock Object`

Properties

- `name string`: The type of block.
- `attributes ?Object`: Attributes to pass to the newly created block.
- `attributesToCopy ?Array<string>`: Attributes to be copied from adjacent blocks when inserted.

[getDraggedBlockClientIds](#)

Returns the client ids of any blocks being directly dragged.

This does not include children of a parent being dragged.

Parameters

- `state Object`: Global application state.

Returns

- `string []`: Array of dragged block client ids.

[getFirstMultiSelectedBlockClientId](#)

Returns the client ID of the first block in the multi-selection set, or null if there is no multi-selection.

Parameters

- *state Object*: Editor state.

Returns

- ?string: First block client ID in the multi-selection set.

[getGlobalBlockCount](#)

Returns the total number of blocks, or the total number of blocks with a specific name in a post. The number returned includes nested blocks.

Parameters

- *state Object*: Global application state.
- *blockName ?string*: Optional block name, if specified only blocks of that type will be counted.

Returns

- *number*: Number of blocks in the post, or number of blocks with name equal to *blockName*.

[getInserterItems](#)

Determines the items that appear in the inserter. Includes both static items (e.g. a regular block type) and dynamic items (e.g. a reusable block).

Each item object contains what's necessary to display a button in the inserter and handle its selection.

The ‘frecency’ property is a heuristic (<https://en.wikipedia.org/wiki/Frecency>) that combines block usage frequency and recency.

Items are returned ordered descendingly by their ‘utility’ and ‘frecency’.

Parameters

- *state Object*: Editor state.
- *rootClientId ?string*: Optional root client ID of block list.

Returns

- *WPEditorInserterItem[]*: Items that appear in inserter.

Type Definition

- *WPEditorInserterItem Object*

Properties

- *id* string: Unique identifier for the item.
- *name* string: The type of block to create.
- *initialAttributes* Object: Attributes to pass to the newly created block.
- *title* string: Title of the item, as it appears in the inserter.
- *icon* string: Dashicon for the item, as it appears in the inserter.
- *category* string: Block category that the item is associated with.
- *keywords* string []: Keywords that can be searched to find this item.
- *isDisabled* boolean: Whether or not the user should be prevented from inserting this item.
- *frecency* number: Heuristic that combines frequency and recency.

[getLastMultiSelectedBlockClientId](#)

Returns the client ID of the last block in the multi-selection set, or null if there is no multi-selection.

Parameters

- *state* Object: Editor state.

Returns

- ?string: Last block client ID in the multi-selection set.

[getLowestCommonAncestorWithSelectedBlock](#)

Given a block client ID, returns the lowest common ancestor with selected client ID.

Parameters

- *state* Object: Editor state.
- *clientId* string: Block from which to find common ancestor client ID.

Returns

- string: Common ancestor client ID or undefined

[getMultiSelectedBlockClientIds](#)

Returns the current multi-selection set of block client IDs, or an empty array if there is no multi-selection.

Parameters

- *state* Object: Editor state.

Returns

- Array: Multi-selected block client IDs.

[getMultiSelectedBlocks](#)

Returns the current multi-selection set of blocks, or an empty array if there is no multi-selection.

Parameters

- *state Object*: Editor state.

Returns

- *Array*: Multi-selected block objects.

[getMultiSelectedBlocksEndClientId](#)

Returns the client ID of the block which ends the multi-selection set, or null if there is no multi-selection.

This is not necessarily the last client ID in the selection.

Related

- [getLastMultiSelectedBlockClientId](#)

Parameters

- *state Object*: Editor state.

Returns

- *?string*: Client ID of block ending multi-selection.

[getMultiSelectedBlocksStartClientId](#)

Returns the client ID of the block which begins the multi-selection set, or null if there is no multi-selection.

This is not necessarily the first client ID in the selection.

Related

- [getFirstMultiSelectedBlockClientId](#)

Parameters

- *state Object*: Editor state.

Returns

- *?string*: Client ID of block beginning multi-selection.

[getNextBlockClientId](#)

Returns the next block's client ID from the given reference start ID. Defaults start to the selected block. Returns null if there is no next block.

Parameters

- *state Object*: Editor state.
- *startClientId ?string*: Optional client ID of block from which to search.

Returns

- *?string*: Adjacent block's client ID, or null if none exists.

[getPatternsByBlockTypes](#)

Returns the list of patterns based on their declared `blockTypes` and a block's name. Patterns can use `blockTypes` to integrate in work flows like suggesting appropriate patterns in a Placeholder state(during insertion) or blocks transformations.

Parameters

- *state Object*: Editor state.
- *blockNames string|string[]*: Block's name or array of block names to find matching patterns.
- *rootClientId ?string*: Optional target root client ID.

Returns

- *Array*: The list of matched block patterns based on declared `blockTypes` and block name.

[getPreviousBlockClientId](#)

Returns the previous block's client ID from the given reference start ID. Defaults start to the selected block. Returns null if there is no previous block.

Parameters

- *state Object*: Editor state.
- *startClientId ?string*: Optional client ID of block from which to search.

Returns

- *?string*: Adjacent block's client ID, or null if none exists.

[getSelectedBlock](#)

Returns the currently selected block, or null if there is no selected block.

Parameters

- *state Object*: Global application state.

Returns

- *?Object*: Selected block.

[getSelectedBlockClientId](#)

Returns the currently selected block client ID, or null if there is no selected block.

Parameters

- *state Object*: Editor state.

Returns

- ?string: Selected block client ID.

[getSelectedBlockClientIds](#)

Returns the current selection set of block client IDs (multiselection or single selection).

Parameters

- *state Object*: Editor state.

Returns

- Array: Multi-selected block client IDs.

[getSelectedBlockCount](#)

Returns the number of blocks currently selected in the post.

Parameters

- *state Object*: Global application state.

Returns

- number: Number of blocks selected in the post.

[getSelectedBlocksInitialCaretPosition](#)

Returns the initial caret position for the selected block. This position is used to position the caret properly when the selected block changes. If the current block is not a RichText, having initial position set to 0 means “focus block”

Parameters

- *state Object*: Global application state.

Returns

- 0 | -1 | null: Initial position.

[getSelectionEnd](#)

Returns the current selection end block client ID, attribute key and text offset.

Parameters

- *state Object*: Block editor state.

Returns

- **WPBlockSelection**: Selection end information.

getSelectionStart

Returns the current selection start block client ID, attribute key and text offset.

Parameters

- *state Object*: Block editor state.

Returns

- **WPBlockSelection**: Selection start information.

getSettings

Returns the editor settings.

Parameters

- *state Object*: Editor state.

Returns

- **Object**: The editor settings object.

getTemplate

Returns the defined block template

Parameters

- *state boolean*:

Returns

- **?Array**: Block Template.

getTemplateLock

Returns the defined block template lock. Optionally accepts a root block client ID as context, otherwise defaulting to the global context.

Parameters

- *state Object*: Editor state.
- *rootClientId ?string*: Optional block root client ID.

Returns

- `string | false`: Block Template Lock

[hasBlockMovingClientId](#)

Returns whether block moving mode is enabled.

Parameters

- `state Object`: Editor state.

Returns

- `string`: Client Id of moving block.

[hasDraggedInnerBlock](#)

Returns true if one of the block's inner blocks is dragged.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.
- `deep boolean`: Perform a deep check.

Returns

- `boolean`: Whether the block has an inner block dragged

[hasInserterItems](#)

Determines whether there are items to show in the inserter.

Parameters

- `state Object`: Editor state.
- `rootClientId ?string`: Optional root client ID of block list.

Returns

- `boolean`: Items that appear in inserter.

[hasMultiSelection](#)

Returns true if a multi-selection has been made, or false otherwise.

Parameters

- `state Object`: Editor state.

Returns

- `boolean`: Whether multi-selection has been made.

[hasSelectedBlock](#)

Returns true if there is a single selected block, or false otherwise.

Parameters

- *state Object*: Editor state.

Returns

- *boolean*: Whether a single block is selected.

[hasSelectedInnerBlock](#)

Returns true if one of the block's inner blocks is selected.

Parameters

- *state Object*: Editor state.
- *clientId string*: Block client ID.
- *deep boolean*: Perform a deep check.

Returns

- *boolean*: Whether the block has an inner block selected

[isAncestorBeingDragged](#)

Returns whether a parent/ancestor of the block is being dragged.

Parameters

- *state Object*: Global application state.
- *clientId string*: Client id for block to check.

Returns

- *boolean*: Whether the block's ancestor is being dragged.

[isAncestorMultiSelected](#)

Returns true if an ancestor of the block is multi-selected, or false otherwise.

Parameters

- *state Object*: Editor state.
- *clientId string*: Block client ID.

Returns

- *boolean*: Whether an ancestor of the block is in multi-selection set.

[isBlockBeingDragged](#)

Returns whether the block is being dragged.

Only returns true if the block is being directly dragged, not if the block is a child of a parent being dragged. See `isAncestorBeingDragged` for child blocks.

Parameters

- `state Object`: Global application state.
- `clientId string`: Client id for block to check.

Returns

- `boolean`: Whether the block is being dragged.

[isBlockHighlighted](#)

Returns true if the current highlighted block matches the block clientId.

Parameters

- `state Object`: Global application state.
- `clientId string`: The block to check.

Returns

- `boolean`: Whether the block is currently highlighted.

[isBlockInsertionPointVisible](#)

Returns true if we should show the block insertion point.

Parameters

- `state Object`: Global application state.

Returns

- `?boolean`: Whether the insertion point is visible or not.

[isBlockMultiSelected](#)

Returns true if the client ID occurs within the block multi-selection, or false otherwise.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `boolean`: Whether block is in multi-selection set.

[isBlockSelected](#)

Returns true if the block corresponding to the specified client ID is currently selected and no multi-selection exists, or false otherwise.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `boolean`: Whether block is selected and multi-selection exists.

[isBlockValid](#)

Returns whether a block is valid or not.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `boolean`: Is Valid.

[isBlockVisible](#)

Tells if the block is visible on the canvas or not.

Parameters

- `state Object`: Global application state.
- `clientId Object`: Client Id of the block.

Returns

- `boolean`: True if the block is visible.

[isBlockWithinSelection](#)

Returns true if the block corresponding to the specified client ID is currently selected but isn't the last of the selected blocks. Here "last" refers to the block sequence in the document, *not* the sequence of multi-selection, which is why `state.selectionEnd` isn't used.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `boolean`: Whether block is selected and not the last in the selection.

[isCaretWithinFormattedText](#)

Deprecated

Returns true if the caret is within formatted text, or false otherwise.

Returns

- `boolean`: Whether the caret is within formatted text.

[isDraggingBlocks](#)

Returns true if the user is dragging blocks, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether user is dragging blocks.

[isFirstMultiSelectedBlock](#)

Returns true if a multi-selection exists, and the block corresponding to the specified client ID is the first block of the multi-selection set, or false otherwise.

Parameters

- `state Object`: Editor state.
- `clientId string`: Block client ID.

Returns

- `boolean`: Whether block is first in multi-selection.

[isGroupable](#)

Indicates if the provided blocks(by client ids) are groupable. We need to have at least one block, have a grouping block name set and be able to remove these blocks.

Parameters

- `state Object`: Global application state.
- `clientIds string []`: Block client ids. If not passed the selected blocks client ids will be used.

Returns

- `boolean`: True if the blocks are groupable.

[isLastBlockChangePersistent](#)

Returns true if the most recent block change is be considered persistent, or false otherwise. A persistent change is one committed by BlockEditorProvider via its `onChange` callback, in addition to `onInput`.

Parameters

- `state Object`: Block editor state.

Returns

- `boolean`: Whether the most recent block change was persistent.

[isMultiSelecting](#)

Whether in the process of multi-selecting or not. This flag is only true while the multi-selection is being selected (by mouse move), and is false once the multi-selection has been settled.

Related

- `hasMultiSelection`

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: True if multi-selecting, false if not.

[isNavigationMode](#)

Returns whether the navigation mode is enabled.

Parameters

- `state Object`: Editor state.

Returns

- `boolean`: Is navigation mode enabled.

[isSelectionEnabled](#)

Selector that returns if multi-selection is enabled or not.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: True if it should be possible to multi-select blocks, false if multi-selection is disabled.

[isTyping](#)

Returns true if the user is typing, or false otherwise.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether user is typing.

[isUngroupable](#)

Indicates if a block is ungroupable. A block is ungroupable if it is a single grouping block with inner blocks. If a block has an `ungroup` transform, it is also ungroupable, without the requirement of being the default grouping block. Additionally a block can only be ungrouped if it has inner blocks and can be removed.

Parameters

- *state Object*: Global application state.
- *clientId string*: Client Id of the block. If not passed the selected block's client id will be used.

Returns

- *boolean*: True if the block is ungroupable.

[isValidTemplate](#)

Returns whether the blocks matches the template or not.

Parameters

- *state boolean*:

Returns

- *?boolean*: Whether the template is valid or not.

[wasBlockJustInserted](#)

Tells if the block with the passed clientId was just inserted.

Parameters

- *state Object*: Global application state.
- *clientId Object*: Client Id of the block.
- *source ?string*: Optional insertion source of the block.

Returns

- *boolean*: True if the block matches the last block inserted from the specified source.

Actions

clearSelectedBlock

Action that clears the block selection.

Returns

- `Object`: Action object.

duplicateBlocks

Action that duplicates a list of blocks.

Parameters

- `clientIds string[]`:
- `updateSelection boolean`:

enterFormattedText

Deprecated

Returns an action object used in signalling that the caret has entered formatted text.

Returns

- `Object`: Action object.

exitFormattedText

Deprecated

Returns an action object used in signalling that the user caret has exited formatted text.

Returns

- `Object`: Action object.

flashBlock

Action that “flashes” the block with a given `clientId` by rhythmically highlighting it.

Parameters

- `clientId string`: Target block client ID.

hideInsertionPoint

Action that hides the insertion point.

[insertAfterBlock](#)

Action that inserts an empty block after a given block.

Parameters

- *clientId* `string`:

[insertBeforeBlock](#)

Action that inserts an empty block before a given block.

Parameters

- *clientId* `string`:

[insertBlock](#)

Action that inserts a single block, optionally at a specific index respective a root block list.

Only allowed blocks are inserted. The action may fail silently for blocks that are not allowed or if a templateLock is active on the block list.

Parameters

- *block Object*: Block object to insert.
- *index ?number*: Index at which block should be inserted.
- *rootClientId ?string*: Optional root client ID of block list on which to insert.
- *updateSelection ?boolean*: If true block selection will be updated. If false, block selection will not change. Defaults to true.
- *meta ?Object*: Optional Meta values to be passed to the action object.

Returns

- `Object`: Action object.

[insertBlocks](#)

Action that inserts an array of blocks, optionally at a specific index respective a root block list.

Only allowed blocks are inserted. The action may fail silently for blocks that are not allowed or if a templateLock is active on the block list.

Parameters

- *blocks Object []*: Block objects to insert.
- *index ?number*: Index at which block should be inserted.
- *rootClientId ?string*: Optional root client ID of block list on which to insert.
- *updateSelection ?boolean*: If true block selection will be updated. If false, block selection will not change. Defaults to true.
- *initialPosition 0 | -1 | null*: Initial focus position. Setting it to null prevent focusing the inserted block.
- *meta ?Object*: Optional Meta values to be passed to the action object.

Returns

- `Object`: Action object.

[insertDefaultBlock](#)

Action that adds a new block of the default type to the block list.

Parameters

- `attributes ?Object`: Optional attributes of the block to assign.
- `rootClientId ?string`: Optional root client ID of block list on which to append.
- `index ?number`: Optional index where to insert the default block.

[mergeBlocks](#)

Action that merges two blocks.

Parameters

- `firstBlockClientId string`: Client ID of the first block to merge.
- `secondBlockClientId string`: Client ID of the second block to merge.

[moveBlocksDown](#)

Undocumented declaration.

[moveBlocksToPosition](#)

Action that moves given blocks to a new position.

Parameters

- `clientIds ?string`: The client IDs of the blocks.
- `fromRootClientId ?string`: Root client ID source.
- `toRootClientId ?string`: Root client ID destination.
- `index number`: The index to move the blocks to.

[moveBlocksUp](#)

Undocumented declaration.

[moveBlockToPosition](#)

Action that moves given block to a new position.

Parameters

- `clientId ?string`: The client ID of the block.
- `fromRootClientId ?string`: Root client ID source.
- `toRootClientId ?string`: Root client ID destination.
- `index number`: The index to move the block to.

[multiSelect](#)

Action that changes block multi-selection.

Parameters

- *start* `string`: First block of the multi selection.
- *end* `string`: Last block of the multiselection.
- *__experimentalInitialPosition* `number | null`: Optional initial position. Pass as null to skip focus within editor canvas.

[receiveBlocks](#)

Deprecated

Returns an action object used in signalling that blocks have been received. Unlike `resetBlocks`, these should be appended to the existing known set, not replacing.

Parameters

- *blocks* `Object []`: Array of block objects.

Returns

- `Object`: Action object.

[registerInserterMediaCategory](#)

Registers a new inserter media category. Once registered, the media category is available in the inserter's media tab.

The following interfaces are used:

Type Definition

- *InserterMediaRequest* `Object`: Interface for inserter media requests.

Properties

- *per_page* `number`: How many items to fetch per page.
- *search* `string`: The search term to use for filtering the results.

Type Definition

- *InserterMediaItem* `Object`: Interface for inserter media responses. Any media resource should map their response to this interface, in order to create the core WordPress media blocks (image, video, audio).

Properties

- *title* `string`: The title of the media item.
- *url* `string`: The source url of the media item.
- *previewUrl* `[string]`: The preview source url of the media item to display in the media list.
- *id* `[number]`: The WordPress id of the media item.

- *sourceId* [number|string]: The id of the media item from external source.
- *alt* [string]: The alt text of the media item.
- *caption* [string]: The caption of the media item.

Usage

```
wp.data.dispatch( 'core/block-editor' ).registerInserterMediaCategory( {
    name: 'openverse',
    labels: {
        name: 'Openverse',
        search_items: 'Search Openverse',
    },
    mediaType: 'image',
    async fetch( query = {} ) {
        const defaultArgs = {
            mature: false,
            excluded_source: 'flickr,inaturalist,wikimedia',
            license: 'pdm,cc0',
        };
        const finalQuery = { ...query, ...defaultArgs };
        // Sometimes you might need to map the supported request params across
        // interface. In this example the `search` query param is named `q`
        const mapFromInserterMediaRequest = {
            per_page: 'page_size',
            search: 'q',
        };
        const url = new URL( 'https://api.openverse.engineering/v1/images/' );
        Object.entries( finalQuery ).forEach( ( [ key, value ] ) => {
            const queryKey = mapFromInserterMediaRequest[ key ] || key;
            url.searchParams.set( queryKey, value );
        } );
        const response = await window.fetch( url, {
            headers: {
                'User-Agent': 'WordPress/inserter-media-fetch',
            },
        } );
        const jsonResponse = await response.json();
        const results = jsonResponse.results;
        return results.map( ( result ) => ( {
            ...result,
            // If your response result includes an `id` prop that you want
            // to be mapped to `InserterMediaItem`'s `sourceId` prop. This can
            // be a report URL getter.
            // Additionally you should always clear the `id` value of your
            // it is used to identify WordPress media items.
            sourceId: result.id,
            id: undefined,
            caption: result.caption,
            previewUrl: result.thumbnail,
        } ) );
    },
    getReportUrl: ( { sourceId } ) =>
        `https://wordpress.org/openverse/image/${ sourceId }/report/`,
}
```

```
    isExternalResource: true,  
} );
```

Parameters

- *category* `InserterMediaCategory`: The inserter media category to register.

Type Definition

- `InserterMediaCategory Object`: Interface for inserter media category.

Properties

- `name string`: The name of the media category, that should be unique among all media categories.
- `labels Object`: Labels for the media category.
- `labels.name string`: General name of the media category. It's used in the inserter media items list.
- `labels.search_items [string]`: Label for searching items. Default is 'Search Posts' / 'Search Pages'.
- `mediaType ('image' | 'audio' | 'video')`: The media type of the media category.
- `fetch (InserterMediaRequest) => Promise<InserterMediaItem[]>`: The function to fetch media items for the category.
- `getReportUrl ((InserterMediaItem) => string)`: If the media category supports reporting media items, this function should return the report url for the media item. It accepts the `InserterMediaItem` as an argument.
- `isExternalResource [boolean]`: If the media category is an external resource, this should be set to true. This is used to avoid making a request to the external resource when the user

removeBlock

Returns an action object used in signalling that the block with the specified client ID is to be removed.

Parameters

- `clientId string`: Client ID of block to remove.
- `selectPrevious boolean`: True if the previous block should be selected when a block is removed.

Returns

- `Object`: Action object.

removeBlocks

Yields action objects used in signalling that the blocks corresponding to the set of specified client IDs are to be removed.

Parameters

- `clientIds string|string[]`: Client IDs of blocks to remove.
- `selectPrevious boolean`: True if the previous block or the immediate parent (if no previous block exists) should be selected when a block is removed.

[replaceBlock](#)

Action that replaces a single block with one or more replacement blocks.

Parameters

- *clientId* (string|string[]): Block client ID to replace.
- *block* (Object|Object[]): Replacement block(s).

Returns

- Object: Action object.

[replaceBlocks](#)

Action that replaces given blocks with one or more replacement blocks.

Parameters

- *clientIds* (string|string[]): Block client ID(s) to replace.
- *blocks* (Object|Object[]): Replacement block(s).
- *indexToSelect* number: Index of replacement block to select.
- *initialPosition* 0|-1|null: Index of caret after in the selected block after the operation.
- *meta* ?Object: Optional Meta values to be passed to the action object.

Returns

- Object: Action object.

[replaceInnerBlocks](#)

Returns an action object used in signalling that the inner blocks with the specified client ID should be replaced.

Parameters

- *rootClientId* string: Client ID of the block whose InnerBlocks will be replaced.
- *blocks* Object[]: Block objects to insert as new InnerBlocks
- *updateSelection* ?boolean: If true block selection will be updated. If false, block selection will not change. Defaults to false.
- *initialPosition* 0|-1|null: Initial block position.

Returns

- Object: Action object.

[resetBlocks](#)

Action that resets blocks state to the specified array of blocks, taking precedence over any other content reflected as an edit in state.

Parameters

- *blocks* Array: Array of blocks.

[resetSelection](#)

Returns an action object used in signalling that selection state should be reset to the specified selection.

Parameters

- *selectionStart* `WPBlockSelection`: The selection start.
- *selectionEnd* `WPBlockSelection`: The selection end.
- *initialPosition* `0|-1|null`: Initial block position.

Returns

- `Object`: Action object.

[selectBlock](#)

Returns an action object used in signalling that the block with the specified client ID has been selected, optionally accepting a position value reflecting its selection directionality. An `initialPosition` of -1 reflects a reverse selection.

Parameters

- *clientId* `string`: Block client ID.
- *initialPosition* `0|-1|null`: Optional initial position. Pass as -1 to reflect reverse selection.

Returns

- `Object`: Action object.

[selectionChange](#)

Action that changes the position of the user caret.

Parameters

- *clientId* `string|WPSelection`: The selected block client ID.
- *attributeKey* `string`: The selected block attribute key.
- *startOffset* `number`: The start offset.
- *endOffset* `number`: The end offset.

Returns

- `Object`: Action object.

[selectNextBlock](#)

Yields action objects used in signalling that the block following the given clientId should be selected.

Parameters

- *clientId* `string`: Block client ID.

[selectPreviousBlock](#)

Yields action objects used in signalling that the block preceding the given clientId (or optionally, its first parent from bottom to top) should be selected.

Parameters

- *clientId* `string`: Block client ID.
- *fallbackToParent* `boolean`: If true, select the first parent if there is no previous block.

[setBlockEditingMode](#)

Sets the block editing mode for a given block.

Related

- [useBlockEditingMode](#)

Parameters

- *clientId* `string`: The block client ID, or '' for the root container.
- *mode* `BlockEditingMode`: The block editing mode. One of 'disabled', 'contentOnly', or 'default'.

Returns

- `Object`: Action object.

[setBlockMovingClientId](#)

Action that enables or disables the block moving mode.

Parameters

- *hasBlockMovingClientId* `string | null`: Enable/Disable block moving mode.

[setBlockVisibility](#)

Action that sets whether given blocks are visible on the canvas.

Parameters

- *updates* `Record<string, boolean>`: For each block's clientId, its new visibility setting.

[setHasControlledInnerBlocks](#)

Action that sets whether a block has controlled inner blocks.

Parameters

- *clientId* `string`: The block's clientId.
- *hasControlledInnerBlocks* `boolean`: True if the block's inner blocks are controlled.

[setNavigationMode](#)

Action that enables or disables the navigation mode.

Parameters

- *isNavigationMode boolean*: Enable/Disable navigation mode.

[setTemplateValidity](#)

Action that resets the template validity.

Parameters

- *isValid boolean*: template validity flag.

Returns

- *Object*: Action object.

[showInsertionPoint](#)

Action that shows the insertion point.

Parameters

- *rootClientId ?string*: Optional root client ID of block list on which to insert.
- *index ?number*: Index at which block should be inserted.
- *_unstableOptions ?Object*: Additional options.

Returns

- *Object*: Action object.

Properties

- *_unstableWithInserter boolean*: Whether or not to show an inserter button.
- *operation WDropOperation*: The operation to perform when applied, either ‘insert’ or ‘replace’ for now.

[startDraggingBlocks](#)

Returns an action object used in signalling that the user has begun to drag blocks.

Parameters

- *clientIds string []*: An array of client ids being dragged

Returns

- *Object*: Action object.

[startMultiSelect](#)

Action that starts block multi-selection.

Returns

- `Object`: Action object.

[startTyping](#)

Returns an action object used in signalling that the user has begun to type.

Returns

- `Object`: Action object.

[stopDraggingBlocks](#)

Returns an action object used in signalling that the user has stopped dragging blocks.

Returns

- `Object`: Action object.

[stopMultiSelect](#)

Action that stops block multi-selection.

Returns

- `Object`: Action object.

[stopTyping](#)

Returns an action object used in signalling that the user has stopped typing.

Returns

- `Object`: Action object.

[synchronizeTemplate](#)

Action that synchronizes the template with the list of blocks.

Returns

- `Object`: Action object.

[toggleBlockHighlight](#)

Action that toggles the highlighted block state.

Parameters

- *clientId* `string`: The block's clientId.
- *isHighlighted* `boolean`: The highlight state.

[toggleBlockMode](#)

Returns an action object used to toggle the block editing mode between visual and HTML modes.

Parameters

- *clientId* `string`: Block client ID.

Returns

- `Object`: Action object.

[toggleSelection](#)

Action that enables or disables block selection.

Parameters

- *isSelectionEnabled* `[boolean]`: Whether block selection should be enabled.

Returns

- `Object`: Action object.

[unsetBlockEditingStyle](#)

Clears the block editing mode for a given block.

Related

- `useBlockEditingStyle`

Parameters

- *clientId* `string`: The block client ID, or '' for the root container.

Returns

- `Object`: Action object.

[updateBlock](#)

Action that updates the block with the specified client ID.

Parameters

- *clientId* `string`: Block client ID.
- *updates* `Object`: Block attributes to be merged.

Returns

- `Object`: Action object.

[updateBlockAttributes](#)

Action that updates attributes of multiple blocks with the specified client IDs.

Parameters

- `clientIds string|string[]`: Block client IDs.
- `attributes Object`: Block attributes to be merged. Should be keyed by clientIDs if `uniqueByBlock` is true.
- `uniqueByBlock boolean`: true if each block in clientIDs array has a unique set of attributes

Returns

- `Object`: Action object.

[updateBlockListSettings](#)

Action that changes the nested settings of a given block.

Parameters

- `clientId string`: Client ID of the block whose nested setting are being received.
- `settings Object`: Object with the new settings for the nested block.

Returns

- `Object`: Action object

[updateSettings](#)

Action that updates the block editor settings.

Parameters

- `settings Object`: Updated settings

Returns

- `Object`: Action object

[validateBlocksToTemplate](#)

Block validity is a function of blocks state (at the point of a reset) and the template setting. As a compromise to its placement across distinct parts of state, it is implemented here as a side effect of the block reset action.

Parameters

- `blocks Array`: Array of blocks.

First published

March 9, 2021

Last updated

January 30, 2024

Edit article

[Improve it on GitHub: The Block Editor's Data](#)

[Previous Block directory](#) [Previous: Block directory](#)

[Next Block Types Data](#) [Next: Block Types Data](#)

Block Types Data

In this article

Table of Contents

- [Selectors](#)
 - [getActiveBlockVariation](#)
 - [getBlockStyles](#)
 - [getBlockSupport](#)
 - [getBlockType](#)
 - [getBlockTypes](#)
 - [getBlockVariations](#)
 - [getCategories](#)
 - [getChildBlockNames](#)
 - [getCollections](#)
 - [getDefaultBlockName](#)
 - [getDefaultBlockVariation](#)
 - [getFreeformFallbackBlockName](#)
 - [getGroupingBlockName](#)
 - [getUnregisteredFallbackBlockName](#)
 - [hasBlockSupport](#)
 - [hasChildBlocks](#)
 - [hasChildBlocksWithInserterSupport](#)
 - [isMatchingSearchTerm](#)
- [Actions](#)
 - [reapplyBlockTypeFilters](#)

[↑ Back to top](#)

Namespace: `core/blocks`.

[Selectors](#)

getActiveBlockVariation

Returns the active block variation for a given block based on its attributes. Variations are determined by their `isActive` property. Which is either an array of block attribute keys or a function.

In case of an array of block attribute keys, the `attributes` are compared to the variation's attributes using strict equality check.

In case of function type, the function should accept a block's attributes and the variation's attributes and determines if a variation is active. A function that accepts a block's attributes and the variation's attributes and determines if a variation is active.

Usage

```
import { __ } from '@wordpress/i18n';
import { store as blocksStore } from '@wordpress/blocks';
import { store as blockEditorStore } from '@wordpress/block-editor';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    // This example assumes that a core/embed block is the first block in
    const activeBlockVariation = useSelect( ( select ) => {
        // Retrieve the list of blocks.
        const [ firstBlock ] = select( blockEditorStore ).getBlocks();

        // Return the active block variation for the first block.
        return select( blocksStore ).getActiveBlockVariation(
            firstBlock.name,
            firstBlock.attributes
        );
    }, [] );

    return activeBlockVariation && activeBlockVariation.name === 'spotify'
        ? (
            <p>{ __( 'Spotify variation' ) }</p>
        )
        : (
            <p>{ __( 'Other variation' ) }</p>
        );
}
```

Parameters

- `state Object`: Data state.
- `blockName string`: Name of block (example: “core/columns”).
- `attributes Object`: Block attributes used to determine active variation.
- `scope [WPBlockVariationScope]`: Block variation scope name.

Returns

- (`WPBlockVariation|undefined`): Active block variation.

getBlockStyles

Returns block styles by block name.

Usage

```
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const buttonBlockStyles = useSelect(
        ( select ) => select( blocksStore ).getBlockStyles( 'core/button' )
        []
    );

    return (
        <ul>
            { buttonBlockStyles &&
                buttonBlockStyles.map( ( style ) => (
                    <li key={ style.name }>{ style.label }</li>
                ) ) }
        </ul>
    );
};


```

Parameters

- *state* Object: Data state.
- *name* string: Block type name.

Returns

- Array?: Block Styles.

[getBlockSupport](#)

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

Usage

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

const ExampleComponent = () => {
    const paragraphBlockSupportValue = useSelect(
        ( select ) =>
            select( blocksStore ).getBlockSupport( 'core/paragraph', 'anchor' )
        []
    );

    return (
        <p>
            { sprintf(
                __( 'core/paragraph supports.anchor value: %s' ),
                paragraphBlockSupportValue
            ) }
        </p>
    );
};


```

```
) ;  
};
```

Parameters

- *state* Object: Data state.
- *nameOrType* (string|Object): Block name or type object
- *feature* Array|string: Feature to retrieve
- *defaultSupports* *: Default value to return if not explicitly defined

Returns

- ?: Block support value

getBlockType

Returns a block type by name.

Usage

```
import { store as blocksStore } from '@wordpress/blocks';  
import { useSelect } from '@wordpress/data';  
  
const ExampleComponent = () => {  
    const paragraphBlock = useSelect(  
        ( select ) => ( select ) =>  
            select( blocksStore ).getBlockType( 'core/paragraph' ),  
        []  
    );  
  
    return (  
        <ul>  
            { paragraphBlock &&  
                Object.entries( paragraphBlock.supports ).map(  
                    ( blockSupportsEntry ) => {  
                        const [ propertyName, value ] = blockSupportsEntry;  
                        return (  
                            <li  
                                key={ propertyName }  
                                >{ `${ propertyName } : ${ value }` }</li>  
                        );  
                    }  
                )  
            )  
        </ul>  
    );  
};
```

Parameters

- *state* Object: Data state.
- *name* string: Block type name.

Returns

- Object?: Block Type.

[getBlockTypes](#)

Returns all the available block types.

Usage

```
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const blockTypes = useSelect(
        ( select ) => select( blocksStore ).getBlockTypes(),
        []
    );

    return (
        <ul>
            { blockTypes.map( ( block ) => (
                <li key={ block.name }>{ block.title }</li>
            ) ) }
        </ul>
    );
};


```

Parameters

- *state* Object: Data state.

Returns

- Array: Block Types.

[getBlockVariations](#)

Returns block variations by block name.

Usage

```
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const socialLinkVariations = useSelect(
        ( select ) =>
            select( blocksStore ).getBlockVariations( 'core/social-link' )
        []
    );

    return (
        <ul>
            { socialLinkVariations &&
                socialLinkVariations.map( ( variation ) => (
                    <li key={ variation.name }>{ variation.title }</li>
                ) ) }
        </ul>
    );
};


```

```
        </ul>
    );
};
```

Parameters

- *state* Object: Data state.
- *blockName* string: Block type name.
- *scope* [WPBlockVariationScope]: Block variation scope name.

Returns

- (WPBlockVariation[] | void): Block variations.

getCategories

Returns all the available block categories.

Usage

```
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const blockCategories = useSelect(
        ( select ) => select( blocksStore ).getCategories(),
        []
    );

    return (
        <ul>
            { blockCategories.map( ( category ) => (
                <li key={ category.slug }>{ category.title }</li>
            ) ) }
        </ul>
    );
};
```

Parameters

- *state* Object: Data state.

Returns

- WPBlockCategory[]: Categories list.

getChildBlockNames

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

Usage

```
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';
```

```
const ExampleComponent = () => {
  const childBlockNames = useSelect(
    ( select ) =>
      select( blocksStore ).getChildBlockNames( 'core/navigation' ),
    []
  );

  return (
    <ul>
      { childBlockNames &&
        childBlockNames.map( ( child ) => (
          <li key={ child }>{ child }</li>
        ) ) }
    </ul>
  );
};
```

Parameters

- *state* Object: Data state.
- *blockName* string: Block type name.

Returns

- Array: Array of child block names.

[getCollections](#)

Returns all the available collections.

Usage

```
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
  const blockCollections = useSelect(
    ( select ) => select( blocksStore ).getCollections(),
    []
  );

  return (
    <ul>
      { Object.values( blockCollections ).length > 0 &&
        Object.values( blockCollections ).map( ( collection ) => (
          <li key={ collection.title }>{ collection.title }</li>
        ) ) }
    </ul>
  );
};
```

Parameters

- *state* Object: Data state.

Returns

- `Object`: Collections list.

[getDefaultValue](#)

Returns the name of the default block name.

Usage

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

const ExampleComponent = () => {
    const defaultBlockName = useSelect(
        ( select ) => select( blocksStore ).getDefaultValue(),
        []
    );

    return (
        defaultBlockName && (
            <p>
                { sprintf( __( 'Default block name: %s' ), defaultBlockName ) }
            </p>
        )
    );
};
```

Parameters

- `state Object`: Data state.

Returns

- `string?`: Default block name.

[getDefaultBlockVariation](#)

Returns the default block variation for the given block type. When there are multiple variations annotated as the default one, the last added item is picked. This simplifies registering overrides. When there is no default variation set, it returns the first item.

Usage

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

const ExampleComponent = () => {
    const defaultEmbedBlockVariation = useSelect(
        ( select ) =>
            select( blocksStore ).getDefaultBlockVariation( 'core/embed' )
    );
};
```

```

);
return (
    defaultEmbedBlockVariation && (
        <p>
            { sprintf(
                __( 'core/embed default variation: %s' ),
                defaultEmbedBlockVariation.title
            ) }
        </p>
    )
);
}

```

Parameters

- *state* Object: Data state.
- *blockName* string: Block type name.
- *scope* [WPBlockVariationScope]: Block variation scope name.

Returns

- ?WPBlockVariation: The default block variation.

getFreeformFallbackBlockName

Returns the name of the block for handling non-block content.

Usage

```

import { __, sprintf } from '@wordpress/i18n';
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const freeformFallbackBlockName = useSelect(
        ( select ) => select( blocksStore ).getFreeformFallbackBlockName()
        []
    );

    return (
        freeformFallbackBlockName && (
            <p>
                { sprintf(
                    __( 'Freeform fallback block name: %s' ),
                    freeformFallbackBlockName
                ) }
            </p>
        )
    );
}

```

Parameters

- *state* Object: Data state.

Returns

- string?: Name of the block for handling non-block content.

[getGroupingBlockName](#)

Returns the name of the block for handling the grouping of blocks.

Usage

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

const ExampleComponent = () => {
    const groupingBlockName = useSelect(
        ( select ) => select( blocksStore ).getGroupingBlockName(),
        []
    );

    return (
        groupingBlockName && (
            <p>
                { sprintf(
                    __( 'Default grouping block name: %s' ),
                    groupingBlockName
                ) }
            </p>
        )
    );
};
```

Parameters

- *state* Object: Data state.

Returns

- string?: Name of the block for handling the grouping of blocks.

[getUnregisteredFallbackBlockName](#)

Returns the name of the block for handling unregistered blocks.

Usage

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

```

const ExampleComponent = () => {
  const unregisteredFallbackBlockName = useSelect(
    ( select ) => select( blocksStore ).getUnregisteredFallbackBlockName()
  );
  return (
    unregisteredFallbackBlockName && (
      <p>
        { sprintf(
          __( 'Unregistered fallback block name: %s' ),
          unregisteredFallbackBlockName
        ) }
      </p>
    )
  );
};

```

Parameters

- *state* Object: Data state.

Returns

- string?: Name of the block for handling unregistered blocks.

[hasBlockSupport](#)

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

Usage

```

import { __, sprintf } from '@wordpress/i18n';
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
  const paragraphBlockSupportClassName = useSelect( ( select ) =>
    select( blocksStore ).hasBlockSupport( 'core/paragraph', 'className' )
  );
  return (
    <p>
      { sprintf(
        __( 'core/paragraph supports custom class name?: %s' ),
        paragraphBlockSupportClassName
      ) }
    </p>
  );
};

```

Parameters

- *state Object*: Data state.
- *nameOrType (string|Object)*: Block name or type object.
- *feature string*: Feature to test.
- *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.

Usage

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

const ExampleComponent = () => {
    const navigationBlockHasChildBlocks = useSelect(
        ( select ) => select( blocksStore ).hasChildBlocks( 'core/navigation' )
    );

    return (
        <p>
            { sprintf(
                __( 'core/navigation has child blocks: %s' ),
                navigationBlockHasChildBlocks
            ) }
        </p>
    );
};
```

Parameters

- *state Object*: Data state.
- *blockName string*: Block type name.

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.

Usage

```

import { __, sprintf } from '@wordpress/i18n';
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const navigationBlockHasChildBlocksWithInserterSupport = useSelect(
        ( select ) =>
            select( blocksStore ).hasChildBlocksWithInserterSupport(
                'core/navigation'
            ),
        []
    );

    return (
        <p>
            { sprintf(
                __(
                    'core/navigation has child blocks with inserter support',
                    navigationBlockHasChildBlocksWithInserterSupport
                )
            ) }
        </p>
    );
};


```

Parameters

- *state Object*: Data state.
- *blockName string*: Block type name.

Returns

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

[isMatchingSearchTerm](#)

Returns true if the block type by the given name or object value matches a search term, or false otherwise.

Usage

```

import { __, sprintf } from '@wordpress/i18n';
import { store as blocksStore } from '@wordpress/blocks';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const termFound = useSelect(
        ( select ) =>
            select( blocksStore ).isMatchingSearchTerm(
                'core/navigation',
                'theme'
            ),
        []
    );
};


```

```

);
return (
<p>
{ sprintf(
  __(
    'Search term was found in the title, keywords, category
  ),
  termFound
) }
</p>
);
}

```

Parameters

- *state Object*: Blocks state.
- *nameOrType (string|Object)*: Block name or type object.
- *searchTerm string*: Search term by which to filter.

Returns

- *Object []*: Whether block type matches search term.

[Actions](#)

The actions in this package shouldn't be used directly. Instead, use the functions listed in the public API [here](#)

[reapplyBlockTypeFilters](#)

Signals that all block types should be computed again. It uses stored unprocessed block types and all the most recent list of registered filters.

It addresses the issue where third party block filters get registered after third party blocks. A sample sequence: 1. Filter A. 2. Block B. 3. Block C. 4. Filter D. 5. Filter E. 6. Block F. 7. Filter G. In this scenario some filters would not get applied for all blocks because they are registered too late.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Block Types Data](#)”

[Previous The Block Editor's Data](#) [Previous: The Block Editor's Data](#)

[Next The Commands Data](#) [Next: The Commands Data](#)

The Commands Data

In this article

Table of Contents

- [Selectors](#)
 - [getCommandLoaders](#)
 - [getCommands](#)
 - [getContext](#)
 - [isOpen](#)
- [Actions](#)
 - [close](#)
 - [open](#)
 - [registerCommand](#)
 - [registerCommandLoader](#)
 - [unregisterCommand](#)
 - [unregisterCommandLoader](#)

[↑ Back to top](#)

Namespace: core/commands.

Selectors

[getCommandLoaders](#)

Returns the registered command loaders.

Parameters

- *state Object*: State tree.
- *contextual boolean*: Whether to return only contextual command loaders.

Returns

- `import('./actions').WPCommandLoaderConfig[]`: The list of registered command loaders.

[getCommands](#)

Returns the registered static commands.

Parameters

- *state Object*: State tree.
- *contextual boolean*: Whether to return only contextual commands.

Returns

- `import('./actions').WPCommandConfig[]`: The list of registered commands.

[getContext](#)

Returns whether the active context.

Parameters

- *state Object*: State tree.

Returns

- *string*: Context.

[isOpen](#)

Returns whether the command palette is open.

Parameters

- *state Object*: State tree.

Returns

- *boolean*: Returns whether the command palette is open.

[Actions](#)

[close](#)

Closes the command palette.

Returns

- *Object*: action.

[open](#)

Opens the command palette.

Returns

- *Object*: action.

[registerCommand](#)

Returns an action object used to register a new command.

Parameters

- *config WPCommandConfig*: Command config.

Returns

- *Object*: action.

[registerCommandLoader](#)

Register command loader.

Parameters

- *config* `WPCommandLoaderConfig`: Command loader config.

Returns

- `Object`: action.

[unregisterCommand](#)

Returns an action object used to unregister a command.

Parameters

- *name* `string`: Command name.

Returns

- `Object`: action.

[unregisterCommandLoader](#)

Unregister command loader hook.

Parameters

- *name* `string`: Command loader name.

Returns

- `Object`: action.

First published

September 6, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: The Commands Data](#)”

[Previous Block Types Data](#) [Previous: Block Types Data](#)

[Next Customize Widgets](#) [Next: Customize Widgets](#)

Customize Widgets

In this article

Table of Contents

- [Selectors](#)
 - [isInserterOpened](#)
- [Actions](#)
 - [setIsInserterOpened](#)

[↑ Back to top](#)

Namespace: core/customize-widgets.

[Selectors](#)

[isInserterOpened](#)

Returns true if the inserter is opened.

Usage

```
import { store as customizeWidgetsStore } from '@wordpress/customize-widgets';
import { __ } from '@wordpress/i18n';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const { isInserterOpened } = useSelect(
        ( select ) => select( customizeWidgetsStore ),
        []
    );

    return isInserterOpened()
        ? __( 'Inserter is open' )
        : __( 'Inserter is closed.' );
};


```

Parameters

- *state* Object: Global application state.

Returns

- boolean: Whether the inserter is opened.

[Actions](#)

[setIsInserterOpened](#)

Returns an action object used to open/close the inserter.

Usage

```
import { useState } from 'react';
import { store as customizeWidgetsStore } from '@wordpress/customize-widgets';
import { __ } from '@wordpress/i18n';
import { useDispatch } from '@wordpress/data';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    const { setIsInserterOpened } = useDispatch( customizeWidgetsStore );
    const [ isOpen, setIsOpen ] = useState( false );

    return (
        <Button
            onClick={ () => {
                setIsInserterOpened( ! isOpen );
                setIsOpen( ! isOpen );
            } }
        >
            { __( 'Open/close inserter' ) }
        </Button>
    );
};


```

Parameters

- ***value* boolean|Object**: Whether the inserter should be opened (true) or closed (false).
To specify an insertion point, use an object.
- ***value.rootClientId* string**: The root client ID to insert at.
- ***value.insertionIndex* number**: The index to insert at.

Returns

- **Action**: Action object.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Customize Widgets”](#)

[Previous](#) [The Commands Data](#) [Previous: The Commands Data](#)

[Next](#) [The Editor’s UI Data](#) [Next: The Editor’s UI Data](#)

The Editor's UI Data

In this article

Table of Contents

- [Selectors](#)

- [areMetaBoxesInitialized](#)
- [getActiveGeneralSidebarName](#)
- [getActiveMetaBoxLocations](#)
- [getAllMetaBoxes](#)
- [getEditedPostTemplate](#)
- [getEditorMode](#)
- [getHiddenBlockTypes](#)
- [getMetaBoxesPerLocation](#)
- [getPreference](#)
- [getPreferences](#)
- [hasMetaBoxes](#)
- [isEditingTemplate](#)
- [isEditorPanelEnabled](#)
- [isEditorPanelOpened](#)
- [isEditorPanelRemoved](#)
- [isEditorSidebarOpened](#)
- [isFeatureActive](#)
- [isInserterOpened](#)
- [isListViewOpened](#)
- [isMetaBoxLocationActive](#)
- [isMetaBoxLocationVisible](#)
- [isModalActive](#)
- [isPluginItemPinned](#)
- [isPluginSidebarOpened](#)
- [isPublishSidebarOpened](#)
- [isSavingMetaBoxes](#)

- [Actions](#)

- [closeGeneralSidebar](#)
- [closeModal](#)
- [closePublishSidebar](#)
- [hideBlockTypes](#)
- [initializeMetaBoxes](#)
- [metaBoxUpdatesFailure](#)
- [metaBoxUpdatesSuccess](#)
- [openGeneralSidebar](#)
- [openModal](#)
- [openPublishSidebar](#)
- [removeEditorPanel](#)
- [requestMetaBoxUpdates](#)
- [setAvailableMetaBoxesPerLocation](#)
- [setIsEditingTemplate](#)
- [setIsInserterOpened](#)
- [setIsListViewOpened](#)
- [showBlockTypes](#)
- [switchEditorMode](#)

- [toggleDistractionFree](#)
- [toggleEditorPanelEnabled](#)
- [toggleEditorPanelOpened](#)
- [toggleFeature](#)
- [togglePinnedPluginItem](#)
- [togglePublishSidebar](#)
- [updatePreferredStyleVariations](#)

[↑ Back to top](#)

Namespace: `core/edit-post`.

Selectors

areMetaBoxesInitialized

Returns true if meta boxes are initialized.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether meta boxes are initialized.

getActiveGeneralSidebarName

Returns the current active general sidebar name, or null if there is no general sidebar active. The active general sidebar is a unique name to identify either an editor or plugin sidebar.

Examples:

- `edit-post/document`
- `my-plugin/insert-image-sidebar`

Parameters

- `state Object`: Global application state.

Returns

- `?string`: Active general sidebar name.

getActiveMetaBoxLocations

Returns an array of active meta box locations.

Parameters

- `state Object`: Post editor state.

Returns

- `string []`: Active meta box locations.

[getAllMetaBoxes](#)

Returns the list of all the available meta boxes.

Parameters

- `state Object`: Global application state.

Returns

- `Array`: List of meta boxes.

[getEditedPostTemplate](#)

Retrieves the template of the currently edited post.

Returns

- `Object ?`: Post Template.

[getEditorMode](#)

Returns the current editing mode.

Parameters

- `state Object`: Global application state.

Returns

- `string`: Editing mode.

[getHiddenBlockTypes](#)

Returns an array of blocks that are hidden.

Returns

- `Array`: A list of the hidden block types

[getMetaBoxesPerLocation](#)

Returns the list of all the available meta boxes for a given location.

Parameters

- `state Object`: Global application state.
- `location string`: Meta box location to test.

Returns

- ?Array: List of meta boxes.

[getPreference](#)

Parameters

- *state Object*: Global application state.
- *preferenceKey string*: Preference Key.
- *defaultValue **: Default Value.

Returns

- *: Preference Value.

[getPreferences](#)

Returns the preferences (these preferences are persisted locally).

Parameters

- *state Object*: Global application state.

Returns

- Object: Preferences Object.

[hasMetaBoxes](#)

Returns true if the post is using Meta Boxes

Parameters

- *state Object*: Global application state

Returns

- boolean: Whether there are metaboxes or not.

[isEditingTemplate](#)

Deprecated

Returns true if the template editing mode is enabled.

[isEditorPanelEnabled](#)

Deprecated

Returns true if the given panel is enabled, or false otherwise. Panels are enabled by default.

Parameters

- *state Object*: Global application state.
- *panelName string*: A string that identifies the panel.

Returns

- *boolean*: Whether or not the panel is enabled.

[isEditorPanelOpened](#)

Deprecated

Returns true if the given panel is open, or false otherwise. Panels are closed by default.

Parameters

- *state Object*: Global application state.
- *panelName string*: A string that identifies the panel.

Returns

- *boolean*: Whether or not the panel is open.

[isEditorPanelRemoved](#)

Deprecated

Returns true if the given panel was programmatically removed, or false otherwise. All panels are not removed by default.

Parameters

- *state Object*: Global application state.
- *panelName string*: A string that identifies the panel.

Returns

- *boolean*: Whether or not the panel is removed.

[isEditorSidebarOpened](#)

Returns true if the editor sidebar is opened.

Parameters

- *state Object*: Global application state

Returns

- *boolean*: Whether the editor sidebar is opened.

[isFeatureActive](#)

Returns whether the given feature is enabled or not.

Parameters

- *state Object*: Global application state.
- *feature string*: Feature slug.

Returns

- *boolean*: Is active.

[isInserterOpened](#)

Deprecated

Returns true if the inserter is opened.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the inserter is opened.

[isListViewOpened](#)

Returns true if the list view is opened.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the list view is opened.

[isMetaBoxLocationActive](#)

Returns true if there is an active meta box in the given location, or false otherwise.

Parameters

- *state Object*: Post editor state.
- *location string*: Meta box location to test.

Returns

- *boolean*: Whether the meta box location is active.

[isMetaBoxLocationVisible](#)

Returns true if a metabox location is active and visible

Parameters

- *state Object*: Post editor state.

- *location* `string`: Meta box location to test.

Returns

- `boolean`: Whether the meta box location is active and visible.

[isModalActive](#)

Deprecated since WP 6.3 use `core/interface` store's selector with the same name instead.

Returns true if a modal is active, or false otherwise.

Parameters

- *state* `Object`: Global application state.
- *modalName* `string`: A string that uniquely identifies the modal.

Returns

- `boolean`: Whether the modal is active.

[isPluginItemPinned](#)

Returns true if the plugin item is pinned to the header. When the value is not set it defaults to true.

Parameters

- *state* `Object`: Global application state.
- *pluginName* `string`: Plugin item name.

Returns

- `boolean`: Whether the plugin item is pinned.

[isPluginSidebarOpened](#)

Returns true if the plugin sidebar is opened.

Parameters

- *state* `Object`: Global application state.

Returns

- `boolean`: Whether the plugin sidebar is opened.

[isPublishSidebarOpened](#)

Returns true if the publish sidebar is opened.

Parameters

- *state* `Object`: Global application state

Returns

- `boolean`: Whether the publish sidebar is open.

[isSavingMetaBoxes](#)

Returns true if the Meta Boxes are being saved.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether the metaboxes are being saved.

[Actions](#)

[closeGeneralSidebar](#)

Returns an action object signalling that the user closed the sidebar.

[closeModal](#)

Deprecated since WP 6.3 use `core/interface` store's action with the same name instead.

Returns an action object signalling that the user closed a modal.

Returns

- `Object`: Action object.

[closePublishSidebar](#)

Returns an action object used in signalling that the user closed the publish sidebar.

Returns

- `Object`: Action object.

[hideBlockTypes](#)

Update the provided block types to be hidden.

Parameters

- `blockNames string []`: Names of block types to hide.

[initializeMetaBoxes](#)

Initializes WordPress `postboxes` script and the logic for saving meta boxes.

[metaBoxUpdatesFailure](#)

Returns an action object used to signal a failed meta box update.

Returns

- `Object`: Action object.

[metaBoxUpdatesSuccess](#)

Returns an action object used to signal a successful meta box update.

Returns

- `Object`: Action object.

[openGeneralSidebar](#)

Returns an action object used in signalling that the user opened an editor sidebar.

Parameters

- `name ?string`: Sidebar name to be opened.

[openModal](#)

Deprecated since WP 6.3 use `core/interface` store's action with the same name instead.

Returns an action object used in signalling that the user opened a modal.

Parameters

- `name string`: A string that uniquely identifies the modal.

Returns

- `Object`: Action object.

[openPublishSidebar](#)

Returns an action object used in signalling that the user opened the publish sidebar.

Returns

- `Object`: Action object

[removeEditorPanel](#)

Deprecated

Returns an action object used to remove a panel from the editor.

Parameters

- *panelName* string: A string that identifies the panel to remove.

Returns

- Object: Action object.

[requestMetaBoxUpdates](#)

Update a metabox.

[setAvailableMetaBoxesPerLocation](#)

Stores info about which Meta boxes are available in which location.

Parameters

- *metaBoxesPerLocation* Object: Meta boxes per location.

[setIsEditingTemplate](#)

Deprecated

Returns an action object used to switch to template editing.

[setIsInserterOpened](#)

Deprecated

Returns an action object used to open/close the inserter.

Parameters

- *value* boolean | Object: Whether the inserter should be opened (true) or closed (false).

[setIsListViewOpened](#)

Deprecated

Returns an action object used to open/close the list view.

Parameters

- *isOpen* boolean: A boolean representing whether the list view should be opened or closed.

[showBlockTypes](#)

Update the provided block types to be visible.

Parameters

- *blockNames* string []: Names of block types to show.

[switchEditorMode](#)

Triggers an action used to switch editor mode.

Parameters

- *mode* string: The editor mode.

[toggleDistractionFree](#)

Action that toggles Distraction free mode. Distraction free mode expects there are no sidebars, as due to the z-index values set, you can't close sidebars.

[toggleEditorPanelEnabled](#)

Deprecated

Returns an action object used to enable or disable a panel in the editor.

Parameters

- *panelName* string: A string that identifies the panel to enable or disable.

Returns

- Object: Action object.

[toggleEditorPanelOpened](#)

Deprecated

Opens a closed panel and closes an open panel.

Parameters

- *panelName* string: A string that identifies the panel to open or close.

[toggleFeature](#)

Triggers an action used to toggle a feature flag.

Parameters

- *feature* string: Feature name.

[togglePinnedPluginItem](#)

Triggers an action object used to toggle a plugin name flag.

Parameters

- *pluginName* string: Plugin name.

[togglePublishSidebar](#)

Returns an action object used in signalling that the user toggles the publish sidebar.

Returns

- `Object`: Action object

[updatePreferredStyleVariations](#)

Returns an action object used in signaling that a style should be auto-applied when a block is created.

Parameters

- `blockName string`: Name of the block.
- `blockStyle ?string`: Name of the style that should be auto applied. If undefined, the “auto apply” setting of the block is removed.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: The Editor’s UI Data](#)“

[Previous](#) [Customize Widgets](#) [Previous: Customize Widgets](#)

[Next](#) [Edit Site](#) [Next: Edit Site](#)

Edit Site

In this article

Table of Contents

- [Selectors](#)
 - [getCanUserCreateMedia](#)
 - [getCurrentTemplateNavigationPanelSubMenu](#)
 - [getCurrentTemplateTemplateParts](#)
 - [getEditedPostContext](#)
 - [getEditedPostId](#)
 - [getEditedPostType](#)
 - [getEditorMode](#)
 - [getHomeTemplateId](#)
 - [getNavigationPanelActiveMenu](#)
 - [getPage](#)
 - [getReusableBlocks](#)
 - [getSettings](#)

- [hasPageContentFocus](#)
- [isFeatureActive](#)
- [isInserterOpened](#)
- [isListViewOpened](#)
- [isNavigationOpened](#)
- [isPage](#)
- [isSaveViewOpened](#)

- **[Actions](#)**

- [addTemplate](#)
- [closeGeneralSidebar](#)
- [openGeneralSidebar](#)
- [openNavigationPanelToMenu](#)
- [removeTemplate](#)
- [revertTemplate](#)
- [setEditedEntity](#)
- [setEditedPostContext](#)
- [setHasPageContentFocus](#)
- [setHomeTemplateId](#)
- [setIsInserterOpened](#)
- [setIsListViewOpened](#)
- [setIsNavigationPanelOpened](#)
- [setIsSaveViewOpened](#)
- [setNavigationMenu](#)
- [setNavigationPanelActiveMenu](#)
- [setPage](#)
- [setTemplate](#)
- [setTemplatePart](#)
- [switchEditorMode](#)
- [toggleDistractionFree](#)
- [toggleFeature](#)
- [updateSettings](#)

[↑ Back to top](#)

Namespace: core/edit-site.

Selectors

getCanUserCreateMedia

Returns whether the current user can create media or not.

Parameters

- *state Object*: Global application state.

Returns

- *Object*: Whether the current user can create media or not.

[getCurrentTemplateNavigationPanelSubMenu](#)

Deprecated

[getCurrentTemplateTemplateParts](#)

Returns the template parts and their blocks for the current edited template.

Parameters

- *state Object*: Global application state.

Returns

- *Array*: Template parts and their blocks in an array.

[getEditedPostContext](#)

Deprecated

Returns the edited post's context object.

Parameters

- *state Object*: Global application state.

Returns

- *Object*: Page.

[getEditedPostId](#)

Returns the ID of the currently edited template or template part.

Parameters

- *state Object*: Global application state.

Returns

- *string?*: Post ID.

[getEditedPostType](#)

Returns the current edited post type (wp_template or wp_template_part).

Parameters

- *state Object*: Global application state.

Returns

- *TemplateType?*: Template type.

[getEditorMode](#)

Returns the current editing mode.

Parameters

- *state Object*: Global application state.

Returns

- *string*: Editing mode.

[getHomeTemplateId](#)

Deprecated

[getNavigationViewActiveMenu](#)

Deprecated

[getPage](#)

Deprecated

Returns the current page object.

Parameters

- *state Object*: Global application state.

Returns

- *Object*: Page.

[getReusableBlocks](#)

Returns any available Reusable blocks.

Parameters

- *state Object*: Global application state.

Returns

- *Array*: The available reusable blocks.

[getSettings](#)

Returns the site editor settings.

Parameters

- *state Object*: Global application state.

Returns

- `Object`: Settings.

[hasPageContentFocus](#)

Deprecated

Whether or not the editor allows only page content to be edited.

Returns

- `boolean`: Whether or not focus is on editing page content.

[isFeatureActive](#)

Deprecated

Returns whether the given feature is enabled or not.

Parameters

- `state Object`: Global application state.
- `featureName string`: Feature slug.

Returns

- `boolean`: Is active.

[isInserterOpened](#)

Deprecated

Returns true if the inserter is opened.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether the inserter is opened.

[isListViewOpened](#)

Returns true if the list view is opened.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether the list view is opened.

[isNavigationOpened](#)

Deprecated

[isPage](#)

Whether or not the editor has a page loaded into it.

Returns

- `setPage`

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether or not the editor has a page loaded into it.

[isSaveViewOpened](#)

Returns the current opened/closed state of the save panel.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: True if the save panel should be open; false if closed.

[Actions](#)

[addTemplate](#)

Deprecated

Action that adds a new template and sets it as the current template.

Parameters

- `template Object`: The template.

Returns

- `Object`: Action object used to set the current template.

[closeGeneralSidebar](#)

Action that closes the sidebar.

[openGeneralSidebar](#)

Action that opens an editor sidebar.

Parameters

- *name* ?string: Sidebar name to be opened.

[openNavigationPanelToMenu](#)

Deprecated

Opens the navigation panel and sets its active menu at the same time.

[removeTemplate](#)

Action that removes a template.

Parameters

- *template Object*: The template object.

[revertTemplate](#)

Reverts a template to its original theme-provided file.

Parameters

- *template Object*: The template to revert.
- *options [Object]*:
- *options.allowUndo [boolean]*: Whether to allow the user to undo reverting the template.
Default true.

[setEditedEntity](#)

Action that sets an edited entity.

Parameters

- *postType string*: The entity's post type.
- *postId string*: The entity's ID.
- *context Object*: The entity's context.

Returns

- *Object*: Action object.

[setEditedPostContext](#)

Set's the current block editor context.

Parameters

- *context Object*: The context object.

Returns

- `Object`: Action object.

[setHasPageContentFocus](#)

Sets whether or not the editor allows only page content to be edited.

Parameters

- `hasPageContentFocus boolean`: True to allow only page content to be edited, false to allow template to be edited.

[setHomeTemplateId](#)

Deprecated

[setIsInserterOpened](#)

Deprecated

Returns an action object used to open/close the inserter.

Parameters

- `value boolean | Object`: Whether the inserter should be opened (true) or closed (false).

[setIsListViewOpened](#)

Deprecated

Returns an action object used to open/close the list view.

Parameters

- `isOpen boolean`: A boolean representing whether the list view should be opened or closed.

[setIsNavigationViewOpened](#)

Deprecated

Sets whether the navigation panel should be open.

[setIsSaveViewOpened](#)

Sets whether the save view panel should be open.

Parameters

- `isOpen boolean`: If true, opens the save view. If false, closes it. It does not toggle the state, but sets it directly.

[setNavigationMenu](#)

Action that sets a navigation menu.

Parameters

- *navigationMenuId* `string`: The Navigation Menu Post ID.

Returns

- `Object`: Action object.

[setNavigationPanelActiveMenu](#)

Deprecated

Action that sets the active navigation panel menu.

Returns

- `Object`: Action object.

[setPage](#)

Deprecated

Resolves the template for a page and displays both. If no path is given, attempts to use the postId to generate a path like `?p=${ postID }`.

Returns

- `number`: The resolved template ID for the page route.

[setTemplate](#)

Action that sets a template, optionally fetching it from REST API.

Returns

- `Object`: Action object.

[setTemplatePart](#)

Action that sets a template part.

Parameters

- *templatePartId* `string`: The template part ID.

Returns

- `Object`: Action object.

[switchEditorMode](#)

Undocumented declaration.

[toggleDistractionFree](#)

Action that toggles Distraction free mode. Distraction free mode expects there are no sidebars, as due to the z-index values set, you can't close sidebars.

[toggleFeature](#)

Dispatches an action that toggles a feature flag.

Parameters

- *featureName* `string`: Feature name.

[updateSettings](#)

Returns an action object used to update the settings.

Parameters

- *settings* `Object`: New settings.

Returns

- `Object`: Action object.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Edit Site”](#)

[Previous The Editor’s UI Data](#) [Previous: The Editor’s UI Data](#)
[Next Edit Widgets](#) [Next: Edit Widgets](#)

Edit Widgets

In this article

Table of Contents

- [Selectors](#)
 - [canInsertBlockInWidgetArea](#)

- [getEditedWidgetAreas](#)
- [getIsWidgetAreaOpen](#)
- [getParentWidgetAreaBlock](#)
- [getReferenceWidgetBlocks](#)
- [getWidget](#)
- [getWidgetAreaForWidgetId](#)
- [getWidgetAreas](#)
- [getWidgets](#)
- [isInserterOpened](#)
- [isListViewOpened](#)
- [isSavingWidgetAreas](#)
- [Actions](#)
 - [closeGeneralSidebar](#)
 - [moveBlockToWidgetArea](#)
 - [persistStubPost](#)
 - [saveEditedWidgetAreas](#)
 - [saveWidgetArea](#)
 - [saveWidgetAreas](#)
 - [setIsInserterOpened](#)
 - [setIsListViewOpened](#)
 - [setIsWidgetAreaOpen](#)
 - [setWidgetAreasOpenState](#)
 - [setWidgetIdForClientId](#)

[↑ Back to top](#)

Namespace: core/edit-widgets.

Selectors

canInsertBlockInWidgetArea

Returns true if a block can be inserted into a widget area.

Parameters

- *state* Array: The open state of the widget areas.
- *blockName* string: The name of the block being inserted.

Returns

- boolean: True if the block can be inserted in a widget area.

getEditedWidgetAreas

Returns all edited widget area entity records.

Returns

- Object []: List of edited widget area entity records.

[getIsWidgetAreaOpen](#)

Gets whether the widget area is opened.

Parameters

- *state* **Array**: The open state of the widget areas.
- *clientId* **string**: The clientId of the widget area.

Returns

- **boolean**: True if the widget area is open.

[getParentWidgetAreaBlock](#)

Given a child client id, returns the parent widget area block.

Parameters

- *clientId* **string**: The client id of a block in a widget area.

Returns

- **WPBlock**: The widget area block.

[getReferenceWidgetBlocks](#)

Returns all blocks representing reference widgets.

Parameters

- *referenceWidgetName* **string**: Optional. If given, only reference widgets with this name will be returned.

Returns

- **Array**: List of all blocks representing reference widgets

[getWidget](#)

Returns API widget data for a particular widget ID.

Parameters

- *id* **number**: Widget ID.

Returns

- **Object**: API widget data for a particular widget ID.

[getWidgetAreaForWidgetId](#)

Returns widgetArea containing a block identify by given widgetId

Parameters

- *widgetId* `string`: The ID of the widget.

Returns

- `Object`: Containing widget area.

[getWidgetAreas](#)

Returns all API widget areas.

Returns

- `Object []`: API List of widget areas.

[getWidgets](#)

Returns all API widgets.

Returns

- `Object []`: API List of widgets.

[isInserterOpened](#)

Returns true if the inserter is opened.

Parameters

- *state* `Object`: Global application state.

Returns

- `boolean`: Whether the inserter is opened.

[isListViewOpened](#)

Returns true if the list view is opened.

Parameters

- *state* `Object`: Global application state.

Returns

- `boolean`: Whether the list view is opened.

[isSavingWidgetAreas](#)

Returns true if any widget area is currently being saved.

Returns

- `boolean`: True if any widget area is currently being saved. False otherwise.

Actions

closeGeneralSidebar

Returns an action object signalling that the user closed the sidebar.

Returns

- `Object`: Action creator.

moveBlockToWidgetArea

Action that handles moving a block between widget areas

Parameters

- `clientId string`: The clientId of the block to move.
- `widgetAreaId string`: The id of the widget area to move the block to.

persistStubPost

Persists a stub post with given ID to core data store. The post is meant to be in-memory only and shouldn't be saved via the API.

Parameters

- `id string`: Post ID.
- `blocks Array`: Blocks the post should consist of.

Returns

- `Object`: The post object.

saveEditedWidgetAreas

Converts all the blocks from edited widget areas into widgets, and submits a batch request to save everything at once.

Creates a snackbar notice on either success or error.

Returns

- `Function`: An action creator.

saveWidgetArea

Converts all the blocks from a widget area specified by ID into widgets, and submits a batch request to save everything at once.

Parameters

- *widgetAreaId* `string`: ID of the widget area to process.

Returns

- `Function`: An action creator.

[saveWidgetAreas](#)

Converts all the blocks from specified widget areas into widgets, and submits a batch request to save everything at once.

Parameters

- *widgetAreas* `Object[]`: Widget areas to save.

Returns

- `Function`: An action creator.

[setIsInserterOpened](#)

Returns an action object used to open/close the inserter.

Parameters

- *value* `boolean | Object`: Whether the inserter should be opened (true) or closed (false).
To specify an insertion point, use an object.
- *value.rootClientId* `string`: The root client ID to insert at.
- *value.insertionIndex* `number`: The index to insert at.

Returns

- `Object`: Action object.

[setIsListViewOpened](#)

Returns an action object used to open/close the list view.

Parameters

- *isOpen* `boolean`: A boolean representing whether the list view should be opened or closed.

Returns

- `Object`: Action object.

[setIsWidgetAreaOpen](#)

Sets the open state of the widget area.

Parameters

- *clientId* string: The clientId of the widget area.
- *isOpen* boolean: Whether the widget area should be opened.

Returns

- Object: Action.

[setWidgetAreasOpenState](#)

Sets the open state of all the widget areas.

Parameters

- *widgetAreasOpenState* Object: The open states of all the widget areas.

Returns

- Object: Action.

[setWidgetIdForClientId](#)

Sets the clientId stored for a particular widgetId.

Parameters

- *clientId* number: Client id.
- *widgetId* number: Widget id.

Returns

- Object: Action.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Edit Widgets](#)

[Previous](#) [Edit Site Previous: Edit Site](#)

[Next](#) [The Post Editor's Data Next: The Post Editor's Data](#)

The Post Editor's Data

In this article

Table of Contents

- [Selectors](#)

- [canInsertBlockType](#)
- [canUserUseUnfilteredHTML](#)
- [didPostSaveRequestFail](#)
- [didPostSaveRequestSucceed](#)
- [getActivePostLock](#)
- [getAdjacentBlockClientId](#)
- [getAutosaveAttribute](#)
- [getBlock](#)
- [getBlockAttributes](#)
- [getBlockCount](#)
- [getBlockHierarchyRootClientId](#)
- [getBlockIndex](#)
- [getBlockInsertionPoint](#)
- [getBlockListSettings](#)
- [getBlockMode](#)
- [getBlockName](#)
- [getBlockOrder](#)
- [getBlockRootClientId](#)
- [getBlocks](#)
- [getBlocksByClientId](#)
- [getBlockSelectionEnd](#)
- [getBlockSelectionStart](#)
- [getClientIdsOfDescendants](#)
- [getClientIdsWithDescendants](#)
- [getCurrentPost](#)
- [getCurrentPostAttribute](#)
- [getCurrentPostId](#)
- [getCurrentPostLastRevisionId](#)
- [getCurrentPostRevisionsCount](#)
- [getCurrentPostType](#)
- [getCurrentTemplateId](#)
- [getDeviceType](#)
- [getEditedPostAttribute](#)
- [getEditedPostContent](#)
- [getEditedPostPreviewLink](#)
- [getEditedPostSlug](#)
- [getEditedPostVisibility](#)
- [getEditorBlocks](#)
- [getEditorSelection](#)
- [getEditorSelectionEnd](#)
- [getEditorSelectionStart](#)
- [getEditorSettings](#)
- [getFirstMultiSelectedBlockClientId](#)
- [getGlobalBlockCount](#)
- [getInserterItems](#)

- [getLastMultiSelectedBlockClientId](#)
- [getMultiSelectedBlockClientIds](#)
- [getMultiSelectedBlocks](#)
- [getMultiSelectedBlocksEndClientId](#)
- [getMultiSelectedBlocksStartClientId](#)
- [getNextBlockClientId](#)
- [getPermalink](#)
- [getPermalinkParts](#)
- [getPostEdits](#)
- [getPostLockUser](#)
- [getPostTypeLabel](#)
- [getPreviousBlockClientId](#)
- [getRenderingMode](#)
- [getSelectedBlock](#)
- [getSelectedBlockClientId](#)
- [getSelectedBlockCount](#)
- [getSelectedBlocksInitialCaretPosition](#)
- [getStateBeforeOptimisticTransaction](#)
- [getSuggestedPostFormat](#)
- [getTemplate](#)
- [getTemplateLock](#)
- [hasChangedContent](#)
- [hasEditorRedo](#)
- [hasEditorUndo](#)
- [hasInserterItems](#)
- [hasMultiSelection](#)
- [hasNonPostEntityChanges](#)
- [hasSelectedBlock](#)
- [hasSelectedInnerBlock](#)
- [inSomeHistory](#)
- [isAncestorMultiSelected](#)
- [isAutosavingPost](#)
- [isBlockInsertionPointVisible](#)
- [isBlockMultiSelected](#)
- [isBlockSelected](#)
- [isBlockValid](#)
- [isBlockWithinSelection](#)
- [isCaretWithinFormattedText](#)
- [isCleanNewPost](#)
- [isCurrentPostPending](#)
- [isCurrentPostPublished](#)
- [isCurrentPostScheduled](#)
- [isDeletingPost](#)
- [isEditedPostAutosaveable](#)
- [isEditedPostBeingScheduled](#)
- [isEditedPostDateFloating](#)
- [isEditedPostDirty](#)
- [isEditedPostEmpty](#)
- [isEditedPostNew](#)
- [isEditedPostPublishable](#)
- [isEditedPostSaveable](#)
- [isEditorPanelEnabled](#)
- [isEditorPanelOpened](#)
- [isEditorPanelRemoved](#)

- [isFirstMultiSelectedBlock](#)
- [isInserterOpened](#)
- [isListViewOpened](#)
- [isMultiSelecting](#)
- [isPermalinkEditable](#)
- [isPostAutosavingLocked](#)
- [isPostLocked](#)
- [isPostLockTakeover](#)
- [isPostSavingLocked](#)
- [isPreviewingPost](#)
- [isPublishingPost](#)
- [isPublishSidebarEnabled](#)
- [isSavingNonPostEntityChanges](#)
- [isSavingPost](#)
- [isSelectionEnabled](#)
- [isTyping](#)
- [isValidTemplate](#)

- [Actions](#)

- [autosave](#)
- [clearSelectedBlock](#)
- [createUndoLevel](#)
- [disablePublishSidebar](#)
- [editPost](#)
- [enablePublishSidebar](#)
- [enterFormattedText](#)
- [exitFormattedText](#)
- [hideInsertionPoint](#)
- [insertBlock](#)
- [insertBlocks](#)
- [insertDefaultBlock](#)
- [lockPostAutosaving](#)
- [lockPostSaving](#)
- [mergeBlocks](#)
- [moveBlocksDown](#)
- [moveBlocksUp](#)
- [moveBlockToPosition](#)
- [multiSelect](#)
- [receiveBlocks](#)
- [redo](#)
- [refreshPost](#)
- [removeBlock](#)
- [removeBlocks](#)
- [removeEditorPanel](#)
- [replaceBlock](#)
- [replaceBlocks](#)
- [resetBlocks](#)
- [resetEditorBlocks](#)
- [resetPost](#)
- [savePost](#)
- [selectBlock](#)
- [setDeviceType](#)
- [setEditedPost](#)
- [setIsInserterOpened](#)
- [setIsListViewOpened](#)

- [setRenderingMode](#)
- [setTemplateValidity](#)
- [setupEditor](#)
- [setupEditorState](#)
- [showInsertionPoint](#)
- [startMultiSelect](#)
- [startTyping](#)
- [stopMultiSelect](#)
- [stopTyping](#)
- [synchronizeTemplate](#)
- [toggleBlockMode](#)
- [toggleEditorPanelEnabled](#)
- [toggleEditorPanelOpened](#)
- [toggleSelection](#)
- [trashPost](#)
- [undo](#)
- [unlockPostAutosaving](#)
- [unlockPostSaving](#)
- [updateBlock](#)
- [updateBlockAttributes](#)
- [updateBlockListSettings](#)
- [updateEditorSettings](#)
- [updatePost](#)
- [updatePostLock](#)

[↑ Back to top](#)

Namespace: `core/editor.`

Selectors

canInsertBlockType

Related

- `canInsertBlockType` in `core/block-editor` store.

canUserUseUnfilteredHTML

Returns whether or not the user has the `unfiltered_html` capability.

Parameters

- `state Object`: Editor state.

Returns

- `boolean`: Whether the user can or can't post unfiltered HTML.

didPostSaveRequestFail

Returns true if a previous post save was attempted but failed, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether the post save failed.

[didPostSaveRequestSucceed](#)

Returns true if a previous post save was attempted successfully, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether the post was saved successfully.

[getActivePostLock](#)

Returns the active post lock.

Parameters

- `state Object`: Global application state.

Returns

- `Object`: The lock object.

[getAdjacentBlockClientId](#)

Related

- `getAdjacentBlockClientId` in core/block-editor store.

[getAutosaveAttribute](#)

Deprecated since 5.6. Callers should use the `getAutosave(postType, postId, userId)` selector from the ‘@wordpress/core-data’ package and access properties on the returned autosave object using `getPostRawValue`.

Returns an attribute value of the current autosave revision for a post, or null if there is no autosave for the post.

Parameters

- `state Object`: Global application state.
- `attributeName string`: Autosave attribute name.

Returns

- `*`: Autosave attribute value.

[getBlock](#)

Related

- `getBlock` in core/block-editor store.

[getBlockAttributes](#)

Related

- `getBlockAttributes` in core/block-editor store.

[getBlockCount](#)

Related

- `getBlockCount` in core/block-editor store.

[getBlockHierarchyRootClientId](#)

Related

- `getBlockHierarchyRootClientId` in core/block-editor store.

[getBlockIndex](#)

Related

- `getBlockIndex` in core/block-editor store.

[getBlockInsertionPoint](#)

Related

- `getBlockInsertionPoint` in core/block-editor store.

[getBlockListSettings](#)

Related

- `getBlockListSettings` in core/block-editor store.

[getBlockMode](#)

Related

- `getBlockMode` in core/block-editor store.

[getBlockName](#)

Related

- `getBlockName` in core/block-editor store.

[getBlockOrder](#)

Related

- `getBlockOrder` in core/block-editor store.

[getBlockRootClientId](#)

Related

- `getBlockRootClientId` in core/block-editor store.

[getBlocks](#)

Related

- `getBlocks` in core/block-editor store.

[getBlocksByClientId](#)

Related

- `getBlocksByClientId` in core/block-editor store.

[getBlockSelectionEnd](#)

Related

- `getBlockSelectionEnd` in core/block-editor store.

[getBlockSelectionStart](#)

Related

- `getBlockSelectionStart` in core/block-editor store.

[getClientIdsOfDescendants](#)

Related

- `getClientIdsOfDescendants` in core/block-editor store.

[getClientIdsWithDescendants](#)

Related

- `getClientIdsWithDescendants` in core/block-editor store.

[getCurrentPost](#)

Returns the post currently being edited in its last known saved state, not including unsaved edits.
Returns an object containing relevant default post values if the post has not yet been saved.

Parameters

- *state Object*: Global application state.

Returns

- *Object*: Post object.

[getCurrentPostAttribute](#)

Returns an attribute value of the saved post.

Parameters

- *state Object*: Global application state.
- *attributeName string*: Post attribute name.

Returns

- *: Post attribute value.

[getCurrentPostId](#)

Returns the ID of the post currently being edited, or null if the post has not yet been saved.

Parameters

- *state Object*: Global application state.

Returns

- ?number: ID of current post.

[getCurrentPostLastRevisionId](#)

Returns the last revision ID of the post currently being edited, or null if the post has no revisions.

Parameters

- *state Object*: Global application state.

Returns

- ?number: ID of the last revision.

[getCurrentPostRevisionsCount](#)

Returns the number of revisions of the post currently being edited.

Parameters

- *state Object*: Global application state.

Returns

- `number`: Number of revisions.

[getCurrentPostType](#)

Returns the post type of the post currently being edited.

Parameters

- `state Object`: Global application state.

Returns

- `string`: Post type.

[getCurrentTemplateId](#)

Returns the template ID currently being rendered/edited

Parameters

- `state Object`: Global application state.

Returns

- `string?:` Template ID.

[getDeviceType](#)

Returns the current editing canvas device type.

Parameters

- `state Object`: Global application state.

Returns

- `string`: Device type.

[getEditedPostAttribute](#)

Returns a single attribute of the post being edited, preferring the unsaved edit if one exists, but falling back to the attribute for the last known saved state of the post.

Parameters

- `state Object`: Global application state.
- `attributeName string`: Post attribute name.

Returns

- `*`: Post attribute value.

[getEditedPostContent](#)

Returns the content of the post being edited.

Parameters

- `state Object`: Global application state.

Returns

- `string`: Post content.

[getEditedPostPreviewLink](#)

Returns the post preview link

Parameters

- `state Object`: Global application state.

Returns

- `string | undefined`: Preview Link.

[getEditedPostSlug](#)

Returns the slug for the post being edited, preferring a manually edited value if one exists, then a sanitized version of the current post title, and finally the post ID.

Parameters

- `state Object`: Editor state.

Returns

- `string`: The current slug to be displayed in the editor

[getEditedPostVisibility](#)

Returns the current visibility of the post being edited, preferring the unsaved value if different than the saved post. The return value is one of “private”, “password”, or “public”.

Parameters

- `state Object`: Global application state.

Returns

- `string`: Post visibility.

[getEditorBlocks](#)

Return the current block list.

Parameters

- *state Object*:

Returns

- **Array**: Block list.

[getEditorSelection](#)

Returns the current selection.

Parameters

- *state Object*:

Returns

- **WPBlockSelection**: The selection end.

[getEditorSelectionEnd](#)

Deprecated since Gutenberg 10.0.0.

Returns the current selection end.

Parameters

- *state Object*:

Returns

- **WPBlockSelection**: The selection end.

[getEditorSelectionStart](#)

Deprecated since Gutenberg 10.0.0.

Returns the current selection start.

Parameters

- *state Object*:

Returns

- **WPBlockSelection**: The selection start.

[getEditorSettings](#)

Returns the post editor settings.

Parameters

- *state Object*: Editor state.

Returns

- `Object`: The editor settings object.

[getFirstMultiSelectedBlockClientId](#)

Related

- `getFirstMultiSelectedBlockClientId` in core/block-editor store.

[getGlobalBlockCount](#)

Related

- `getGlobalBlockCount` in core/block-editor store.

[getInserterItems](#)

Related

- `getInserterItems` in core/block-editor store.

[getLastMultiSelectedBlockClientId](#)

Related

- `getLastMultiSelectedBlockClientId` in core/block-editor store.

[getMultiSelectedBlockClientIds](#)

Related

- `getMultiSelectedBlockClientIds` in core/block-editor store.

[getMultiSelectedBlocks](#)

Related

- `getMultiSelectedBlocks` in core/block-editor store.

[getMultiSelectedBlocksEndClientId](#)

Related

- `getMultiSelectedBlocksEndClientId` in core/block-editor store.

[getMultiSelectedBlocksStartClientId](#)

Related

- `getMultiSelectedBlocksStartClientId` in core/block-editor store.

[getNextBlockClientId](#)

Returns

- `getNextBlockClientId` in core/block-editor store.

[getPermalink](#)

Returns the permalink for the post.

Parameters

- `state Object`: Editor state.

Returns

- `?string`: The permalink, or null if the post is not viewable.

[getPermalinkParts](#)

Returns the permalink for a post, split into it's three parts: the prefix, the `postName`, and the suffix.

Parameters

- `state Object`: Editor state.

Returns

- `Object`: An object containing the prefix, `postName`, and suffix for the permalink, or null if the post is not viewable.

[getPostEdits](#)

Returns any post values which have been changed in the editor but not yet been saved.

Parameters

- `state Object`: Global application state.

Returns

- `Object`: Object of key value pairs comprising unsaved edits.

[getPostLockUser](#)

Returns details about the post lock user.

Parameters

- `state Object`: Global application state.

Returns

- `Object`: A user object.

[getPostTypeLabel](#)

Returns a post type label depending on the current post.

Parameters

- `state Object`: Global application state.

Returns

- `string | undefined`: The post type label if available, otherwise undefined.

[getPreviousBlockClientId](#)

Related

- `getPreviousBlockClientId` in core/block-editor store.

[getRenderingMode](#)

Returns the post editor's rendering mode.

Parameters

- `state Object`: Editor state.

Returns

- `string`: Rendering mode.

[getSelectedBlock](#)

Related

- `getSelectedBlock` in core/block-editor store.

[getSelectedBlockClientId](#)

Related

- `getSelectedBlockClientId` in core/block-editor store.

[getSelectedBlockCount](#)

Related

- `getSelectedBlockCount` in core/block-editor store.

[getSelectedBlocksInitialCaretPosition](#)

Related

- `getSelectedBlocksInitialCaretPosition` in core/block-editor store.

[getStateBeforeOptimisticTransaction](#)

Deprecated since Gutenberg 9.7.0.

Returns state object prior to a specified optimist transaction ID, or `null` if the transaction corresponding to the given ID cannot be found.

[getSuggestedPostFormat](#)

Returns a suggested post format for the current post, inferred only if there is a single block within the post and it is of a type known to match a default post format. Returns null if the format cannot be determined.

Returns

- `?string`: Suggested post format.

[getTemplate](#)

Related

- `getTemplate` in core/block-editor store.

[getTemplateLock](#)

Related

- `getTemplateLock` in core/block-editor store.

[hasChangedContent](#)

Returns true if content includes unsaved changes, or false otherwise.

Parameters

- `state Object`: Editor state.

Returns

- `boolean`: Whether content includes unsaved changes.

[hasEditorRedo](#)

Returns true if any future editor history snapshots exist, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether redo history exists.

[hasEditorUndo](#)

Returns true if any past editor history snapshots exist, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether undo history exists.

[hasInserterItems](#)

Related

- `hasInserterItems` in core/block-editor store.

[hasMultiSelection](#)

Related

- `hasMultiSelection` in core/block-editor store.

[hasNonPostEntityChanges](#)

Returns true if there are unsaved edits for entities other than the editor's post, and false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether there are edits or not.

[hasSelectedBlock](#)

Related

- `hasSelectedBlock` in core/block-editor store.

[hasSelectedInnerBlock](#)

Related

- `hasSelectedInnerBlock` in core/block-editor store.

[inSomeHistory](#)

Deprecated since Gutenberg 9.7.0.

Returns true if an optimistic transaction is pending commit, for which the before state satisfies the given predicate function.

[isAncestorMultiSelected](#)

Related

- `isAncestorMultiSelected` in core/block-editor store.

[isAutosavingPost](#)

Returns true if the post is autosaving, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether the post is autosaving.

[isBlockInsertionPointVisible](#)

Related

- `isBlockInsertionPointVisible` in core/block-editor store.

[isBlockMultiSelected](#)

Related

- `isBlockMultiSelected` in core/block-editor store.

[isBlockSelected](#)

Related

- `isBlockSelected` in core/block-editor store.

[isBlockValid](#)

Related

- `isBlockValid` in core/block-editor store.

[isBlockWithinSelection](#)

Related

- `isBlockWithinSelection` in core/block-editor store.

[isCaretWithinFormattedText](#)

Related

- `isCaretWithinFormattedText` in core/block-editor store.

[isCleanNewPost](#)

Returns true if there are no unsaved values for the current edit session and if the currently edited post is new (has never been saved before).

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether new post and unsaved values exist.

[isCurrentPostPending](#)

Returns true if post is pending review.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether current post is pending review.

[isCurrentPostPublished](#)

Return true if the current post has already been published.

Parameters

- *state Object*: Global application state.
- *currentPost Object?*: Explicit current post for bypassing registry selector.

Returns

- *boolean*: Whether the post has been published.

[isCurrentPostScheduled](#)

Returns true if post is already scheduled.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether current post is scheduled to be posted.

[isDeletingPost](#)

Returns true if the post is currently being deleted, or false otherwise.

Parameters

- *state Object*: Editor state.

Returns

- *boolean*: Whether post is being deleted.

[isEditedPostAutosaveable](#)

Returns true if the post can be autosaved, or false otherwise.

Parameters

- *state Object*: Global application state.
- *autosave Object*: A raw autosave object from the REST API.

Returns

- *boolean*: Whether the post can be autosaved.

[isEditedPostBeingScheduled](#)

Return true if the post being edited is being scheduled. Preferring the unsaved status values.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the post has been published.

[isEditedPostDateFloating](#)

Returns whether the current post should be considered to have a “floating” date (i.e. that it would publish “Immediately” rather than at a set time).

Unlike in the PHP backend, the REST API returns a full date string for posts where the 0000-00-00T00:00:00 placeholder is present in the database. To infer that a post is set to publish “Immediately” we check whether the date and modified date are the same.

Parameters

- *state Object*: Editor state.

Returns

- *boolean*: Whether the edited post has a floating date value.

[isEditedPostDirty](#)

Returns true if there are unsaved values for the current edit session, or false if the editing state matches the saved or new post.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether unsaved values exist.

[isEditedPostEmpty](#)

Returns true if the edited post has content. A post has content if it has at least one saveable block or otherwise has a non-empty content property assigned.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether post has content.

[isEditedPostNew](#)

Returns true if the currently edited post is yet to be saved, or false if the post has been saved.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the post is new.

[isEditedPostPublishable](#)

Return true if the post being edited can be published.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the post can be published.

[isEditedPostSaveable](#)

Returns true if the post can be saved, or false otherwise. A post must contain a title, an excerpt, or non-empty content to be valid for save.

Parameters

- *state Object*: Global application state.

Returns

- `boolean`: Whether the post can be saved.

[isEditorPanelEnabled](#)

Returns true if the given panel is enabled, or false otherwise. Panels are enabled by default.

Parameters

- `state Object`: Global application state.
- `panelName string`: A string that identifies the panel.

Returns

- `boolean`: Whether or not the panel is enabled.

[isEditorPanelOpened](#)

Returns true if the given panel is open, or false otherwise. Panels are closed by default.

Parameters

- `state Object`: Global application state.
- `panelName string`: A string that identifies the panel.

Returns

- `boolean`: Whether or not the panel is open.

[isEditorPanelRemoved](#)

Returns true if the given panel was programmatically removed, or false otherwise. All panels are not removed by default.

Parameters

- `state Object`: Global application state.
- `panelName string`: A string that identifies the panel.

Returns

- `boolean`: Whether or not the panel is removed.

[isFirstMultiSelectedBlock](#)

Related

- `isFirstMultiSelectedBlock` in `core/block-editor` store.

[isInserterOpened](#)

Returns true if the inserter is opened.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the inserter is opened.

[isListViewOpened](#)

Returns true if the list view is opened.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the list view is opened.

[isMultiSelecting](#)

Related

- *isMultiSelecting* in core/block-editor store.

[isPermalinkEditable](#)

Returns whether the permalink is editable or not.

Parameters

- *state Object*: Editor state.

Returns

- *boolean*: Whether or not the permalink is editable.

[isPostAutosavingLocked](#)

Returns whether post autosaving is locked.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Is locked.

[isPostLocked](#)

Returns whether the post is locked.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Is locked.

[isPostLockTakeover](#)

Returns whether the edition of the post has been taken over.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Is post lock takeover.

[isPostSavingLocked](#)

Returns whether post saving is locked.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Is locked.

[isPreviewingPost](#)

Returns true if the post is being previewed, or false otherwise.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether the post is being previewed.

[isPublishingPost](#)

Returns true if the post is being published, or false otherwise.

Parameters

- *state Object*: Global application state.

Returns

- *boolean*: Whether post is being published.

[isPublishSidebarEnabled](#)

Returns whether the pre-publish panel should be shown or skipped when the user clicks the “publish” button.

Returns

- `boolean`: Whether the pre-publish panel should be shown or not.

[isSavingNonPostEntityChanges](#)

Returns true if non-post entities are currently being saved, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether non-post entities are being saved.

[isSavingPost](#)

Returns true if the post is currently being saved, or false otherwise.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether post is being saved.

[isSelectionEnabled](#)

Related

- `isSelectionEnabled` in core/block-editor store.

[isTyping](#)

Related

- `isTyping` in core/block-editor store.

[isValidTemplate](#)

Related

- `isValidTemplate` in core/block-editor store.

[Actions](#)

[autosave](#)

Action that autosaves the current post. This includes server-side autosaving (default) and client-side (a.k.a. local) autosaving (e.g. on the Web, the post might be committed to Session Storage).

Parameters

- *options Object?*: Extra flags to identify the autosave.

[clearSelectedBlock](#)

Related

- clearSelectedBlock in core/block-editor store.

[createUndoLevel](#)

Deprecated Since WordPress 6.0

Action that creates an undo history record.

[disablePublishSidebar](#)

Disables the publish sidebar.

[editPost](#)

Returns an action object used in signalling that attributes of the post have been edited.

Parameters

- *edits Object*: Post attributes to edit.
- *options Object*: Options for the edit.

[enablePublishSidebar](#)

Enable the publish sidebar.

[enterFormattedText](#)

Related

- enterFormattedText in core/block-editor store.

[exitFormattedText](#)

Related

- exitFormattedText in core/block-editor store.

[hideInsertionPoint](#)

Related

- [hideInsertionPoint](#) in core/block-editor store.

[insertBlock](#)

Related

- [insertBlock](#) in core/block-editor store.

[insertBlocks](#)

Related

- [insertBlocks](#) in core/block-editor store.

[insertDefaultBlock](#)

Related

- [insertDefaultBlock](#) in core/block-editor store.

[lockPostAutosaving](#)

Action that locks post autosaving.

Usage

```
// Lock post autosaving with the lock key `mylock`:  
wp.data.dispatch( 'core/editor' ).lockPostAutosaving( 'mylock' );
```

Parameters

- *lockName* string: The lock name.

Returns

- Object: Action object

[lockPostSaving](#)

Action that locks post saving.

Usage

```
const { subscribe } = wp.data;  
  
const initialPostStatus = wp.data.select( 'core/editor' ).getEditedPostAtt  
  
// Only allow publishing posts that are set to a future date.  
if ( 'publish' !== initialPostStatus ) {
```

```

// Track locking.
let locked = false;

// Watch for the publish event.
let unsubscribe = subscribe( () => {
    const currentPostStatus = wp.data.select( 'core/editor' ).getEditorPostStatus();
    if ( 'publish' !== currentPostStatus ) {

        // Compare the post date to the current date, lock the post if it's future.
        const postDate = new Date( wp.data.select( 'core/editor' ).getEditorPostDate() );
        const currentDate = new Date();
        if ( postDate.getTime() <= currentDate.getTime() ) {
            if ( !locked ) {
                locked = true;
                wp.data.dispatch( 'core/editor' ).lockPostSaving( 'future' );
            }
        } else {
            if ( locked ) {
                locked = false;
                wp.data.dispatch( 'core/editor' ).unlockPostSaving( 'future' );
            }
        }
    }
} );
}

```

Parameters

- *lockName* `string`: The lock name.

Returns

- `Object`: Action object

[mergeBlocks](#)

Related

- `mergeBlocks` in core/block-editor store.

[moveBlocksDown](#)

Related

- `moveBlocksDown` in core/block-editor store.

[moveBlocksUp](#)

Related

- `moveBlocksUp` in core/block-editor store.

[moveBlockToPosition](#)

Related

- moveBlockToPosition in core/block-editor store.

[multiSelect](#)

Related

- multiSelect in core/block-editor store.

[receiveBlocks](#)

Related

- receiveBlocks in core/block-editor store.

[redo](#)

Action that restores last popped state in undo history.

[refreshPost](#)

Deprecated Since WordPress 6.0.

Action for refreshing the current post.

[removeBlock](#)

Related

- removeBlock in core/block-editor store.

[removeBlocks](#)

Related

- removeBlocks in core/block-editor store.

[removeEditorPanel](#)

Returns an action object used to remove a panel from the editor.

Parameters

- *panelName* `string`: A string that identifies the panel to remove.

Returns

- `Object`: Action object.

[replaceBlock](#)

Related

- [replaceBlock](#) in core/block-editor store.

[replaceBlocks](#)

Related

- [replaceBlocks](#) in core/block-editor store.

[resetBlocks](#)

Related

- [resetBlocks](#) in core/block-editor store.

[resetEditorBlocks](#)

Returns an action object used to signal that the blocks have been updated.

Parameters

- *blocks* Array: Block Array.
- *options* ?Object: Optional options.

[resetPost](#)

Deprecated Since WordPress 6.0.

Returns an action object used in signalling that the latest version of the post has been received, either by initialization or save.

[savePost](#)

Action for saving the current post in the editor.

Parameters

- *options* Object:

[selectBlock](#)

Related

- [selectBlock](#) in core/block-editor store.

[setDeviceType](#)

Action that changes the width of the editing canvas.

Parameters

- *deviceType* string:

Returns

- Object: Action object.

[setEditedPost](#)

Returns an action that sets the current post Type and post ID.

Parameters

- *postType* string: Post Type.
- *postId* string: Post ID.

Returns

- Object: Action object.

[setIsInserterOpened](#)

Returns an action object used to open/close the inserter.

Parameters

- *value* boolean | Object: Whether the inserter should be opened (true) or closed (false).
To specify an insertion point, use an object.
- *value.rootClientId* string: The root client ID to insert at.
- *value.insertionIndex* number: The index to insert at.

Returns

- Object: Action object.

[setIsListViewOpened](#)

Returns an action object used to open/close the list view.

Parameters

- *isOpen* boolean: A boolean representing whether the list view should be opened or closed.

Returns

- Object: Action object.

[setRenderingMode](#)

Returns an action used to set the rendering mode of the post editor. We support multiple rendering modes:

- **all**: This is the default mode. It renders the post editor with all the features available. If a template is provided, it's preferred over the post.
- **post-only**: This mode extracts the post blocks from the template and renders only those. The idea is to allow the user to edit the post/page in isolation without the wrapping template.
- **template-locked**: This mode renders both the template and the post blocks but the template blocks are locked and can't be edited. The post blocks are editable.

Parameters

- *mode* `string`: Mode (one of ‘post-only’, ‘template-locked’ or ‘all’).

[setTemplateValidity](#)

Related

- `setTemplateValidity` in core/block-editor store.

[setupEditor](#)

Returns an action generator used in signalling that editor has initialized with the specified post object and editor settings.

Parameters

- *post Object*: Post object.
- *edits Object*: Initial edited attributes object.
- *template Array?*: Block Template.

[setupEditorState](#)

Deprecated

Setup the editor state.

Parameters

- *post Object*: Post object.

[showInsertionPoint](#)

Related

- `showInsertionPoint` in core/block-editor store.

[startMultiSelect](#)

Related

- [startMultiSelect](#) in core/block-editor store.

[startTyping](#)

Related

- [startTyping](#) in core/block-editor store.

[stopMultiSelect](#)

Related

- [stopMultiSelect](#) in core/block-editor store.

[stopTyping](#)

Related

- [stopTyping](#) in core/block-editor store.

[synchronizeTemplate](#)

Related

- [synchronizeTemplate](#) in core/block-editor store.

[toggleBlockMode](#)

Related

- [toggleBlockMode](#) in core/block-editor store.

[toggleEditorPanelEnabled](#)

Returns an action object used to enable or disable a panel in the editor.

Parameters

- *panelName* `string`: A string that identifies the panel to enable or disable.

Returns

- `Object`: Action object.

[toggleEditorPanelOpened](#)

Opens a closed panel and closes an open panel.

Parameters

- *panelName* string: A string that identifies the panel to open or close.

toggleSelection

Related

- toggleSelection in core/block-editor store.

trashPost

Action for trashing the current post in the editor.

undo

Action that pops a record from undo history and undoes the edit.

unlockPostAutosaving

Action that unlocks post autosaving.

Usage

```
// Unlock post saving with the lock key `mylock`:  
wp.data.dispatch( 'core/editor' ).unlockPostAutosaving( 'mylock' );
```

Parameters

- *lockName* string: The lock name.

Returns

- Object: Action object

unlockPostSaving

Action that unlocks post saving.

Usage

```
// Unlock post saving with the lock key `mylock`:  
wp.data.dispatch( 'core/editor' ).unlockPostSaving( 'mylock' );
```

Parameters

- *lockName* string: The lock name.

Returns

- Object: Action object

[updateBlock](#)

Related

- updateBlock in core/block-editor store.

[updateBlockAttributes](#)

Related

- updateBlockAttributes in core/block-editor store.

[updateBlockListSettings](#)

Related

- updateBlockListSettings in core/block-editor store.

[updateEditorSettings](#)

Undocumented declaration.

[updatePost](#)

Deprecated since Gutenberg 9.7.0.

Returns an action object used in signalling that a patch of updates for the latest version of the post have been received.

Returns

- `Object`: Action object.

[updatePostLock](#)

Action that locks the editor.

Parameters

- `lock Object`: Details about the post lock status, user, and nonce.

Returns

- `Object`: Action object.

First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: The Post Editor's Data”](#)

[Previous Edit Widgets](#) [Previous: Edit Widgets](#)

[Next The Keyboard Shortcuts Data](#) [Next: The Keyboard Shortcuts Data](#)

The Keyboard Shortcuts Data

In this article

[Table of Contents](#)

- [Selectors](#)
 - [getAllShortcutKeyCombinations](#)
 - [getAllShortcutRawKeyCombinations](#)
 - [getCategoryShortcuts](#)
 - [getShortcutAliases](#)
 - [getShortcutDescription](#)
 - [getShortcutKeyCombination](#)
 - [getShortcutRepresentation](#)
- [Actions](#)
 - [registerShortcut](#)
 - [unregisterShortcut](#)

[↑ Back to top](#)

Namespace: core/keyboard-shortcuts.

Selectors

[getAllShortcutKeyCombinations](#)

Returns the shortcuts that include aliases for a given shortcut name.

Usage

```
import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect } from '@wordpress/data';
import { createElement } from '@wordpress/element';
import { sprintf } from '@wordpress/i18n';

const ExampleComponent = () => {
    const allShortcutKeyCombinations = useSelect(
        ( select ) =>
            select( keyboardShortcutsStore ).getAllShortcutKeyCombinations(
                'core/edit-post/next-region'
            ),
        []
    );
    return (

```

```

        allShortcutKeyCombinations.length > 0 && (
            <ul>
                { allShortcutKeyCombinations.map(
                    ( { character, modifier }, index ) => (
                        <li key={ index }>
                            { createInterpolateElement(
                                sprintf(
                                    'Character: <code>%s</code> / Modifier',
                                    character,
                                    modifier
                                ),
                                {
                                    code: <code />,
                                }
                            ) )
                        </li>
                    )
                ) }
            </ul>
        )
    );
}

```

Parameters

- `state Object`: Global state.
- `name string`: Shortcut name.

Returns

- `WPShortcutKeyCombination []`: Key combinations.

[getAllShortcutRawKeyCombinations](#)

Returns the raw representation of all the keyboard combinations of a given shortcut name.

Usage

```

import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect } from '@wordpress/data';
import { createInterpolateElement } from '@wordpress/element';
import { sprintf } from '@wordpress/i18n';

const ExampleComponent = () => {
    const allShortcutRawKeyCombinations = useSelect(
        ( select ) =>
            select( keyboardShortcutsStore ).getAllShortcutRawKeyCombinations(
                'core/edit-post/next-region'
            ),
        []
    );
    return (
        allShortcutRawKeyCombinations.length > 0 && (

```

```

        <ul>
            { allShortcutRawKeyCombinations.map(
                ( shortcutRawKeyCombination, index ) => (
                    <li key={ index }>
                        { createInterpolateElement(
                            sprintf(
                                ' <code>%s</code>',
                                shortcutRawKeyCombination
                            ),
                            {
                                code: <code />,
                            }
                        )
                    </li>
                )
            )
        )
    );
}

```

Parameters

- `state Object`: Global state.
- `name string`: Shortcut name.

Returns

- `string []`: Shortcuts.

[getCategoryShortcuts](#)

Returns the shortcut names list for a given category name.

Usage

```

import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const categoryShortcuts = useSelect(
        ( select ) =>
            select( keyboardShortcutsStore ).getCategoryShortcuts( 'block' )
    );
}

return (
    categoryShortcuts.length > 0 && (
        <ul>
            { categoryShortcuts.map( ( categoryShortcut ) => (
                <li key={ categoryShortcut }>{ categoryShortcut }</li>
            ) ) }
        </ul>
    )
)

```

```
) ;  
};
```

Parameters

- *state Object*: Global state.
- *name string*: Category name.

Returns

- *string []*: Shortcut names.

[getShortcutAliases](#)

Returns the aliases for a given shortcut name.

Usage

```
import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect } from '@wordpress/data';
import { createElement } from '@wordpress/element';
import { sprintf } from '@wordpress/i18n';
const ExampleComponent = () => {
    const shortcutAliases = useSelect(
        ( select ) =>
            select( keyboardShortcutsStore ).getShortcutAliases(
                'core/edit-post/next-region'
            ),
        []
    );
    return (
        shortcutAliases.length > 0 && (
            <ul>
                { shortcutAliases.map( ( { character, modifier }, index ) =>
                    <li key={ index }>
                        { createElement(
                            sprintf(
                                'Character: <code>%s</code> / Modifier: <code>%s</code>',
                                character,
                                modifier
                            )
                        );
                    {
                        code: <code />
                    }
                ) }
            </li>
        ) )
    );
};
```

Parameters

- *state* Object: Global state.
- *name* string: Shortcut name.

Returns

- WPShortcutKeyCombination []: Key combinations.

[getShortcutDescription](#)

Returns the shortcut description given its name.

Usage

```
import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect } from '@wordpress/data';
import { __ } from '@wordpress/i18n';
const ExampleComponent = () => {
    const shortcutDescription = useSelect(
        ( select ) =>
            select( keyboardShortcutsStore ).getShortcutDescription(
                'core/edit-post/next-region'
            ),
        []
    );
    return shortcutDescription ? (
        <div>{ shortcutDescription }</div>
    ) : (
        <div>{ __( 'No description.' ) }</div>
    );
};
```

Parameters

- *state* Object: Global state.
- *name* string: Shortcut name.

Returns

- string?: Shortcut description.

[getShortcutKeyCombination](#)

Returns the main key combination for a given shortcut name.

Usage

```
import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect } from '@wordpress/data';
import { createElement } from '@wordpress/element';
import { sprintf } from '@wordpress/i18n';
const ExampleComponent = () => {
```

```

const { character, modifier } = useSelect(
  ( select ) =>
    select( keyboardShortcutsStore ).getShortcutKeyCombination(
      'core/edit-post/next-region'
    ),
  []
);
return (
  <div>
    { createInterpolateElement(
      sprintf(
        'Character: <code>%s</code> / Modifier: <code>%s</code>',
        character,
        modifier
      ),
      {
        code: <code />,
      }
    ) }
  </div>
);
}

```

Parameters

- *state* Object: Global state.
- *name* string: Shortcut name.

Returns

- WPShortcutKeyCombination?: Key combination.

getShortcutRepresentation

Returns a string representing the main key combination for a given shortcut name.

Usage

```

import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect } from '@wordpress/data';
import { sprintf } from '@wordpress/i18n';

const ExampleComponent = () => {
  const { display, raw, ariaLabel } = useSelect( ( select ) => {
    return {
      display: select( keyboardShortcutsStore ).getShortcutRepresentation(
        'core/edit-post/next-region'
      ),
      raw: select( keyboardShortcutsStore ).getShortcutRepresentation(
        'core/edit-post/next-region',
        'raw'
      ),
      ariaLabel: select(

```

```

        keyboardShortcutsStore
    ).getShortcutRepresentation(
        'core/edit-post/next-region',
        'ariaLabel'
    ),
},
],
[]);

return (
<ul>
    <li>{ sprintf( 'display string: %s', display ) }</li>
    <li>{ sprintf( 'raw string: %s', raw ) }</li>
    <li>{ sprintf( 'ariaLabel string: %s', ariaLabel ) }</li>
</ul>
);
};

```

Parameters

- *state Object*: Global state.
- *name string*: Shortcut name.
- *representation keyof FORMATTING_METHODS*: Type of representation (display, raw, ariaLabel).

Returns

- *string?*: Shortcut representation.

Actions

registerShortcut

Returns an action object used to register a new keyboard shortcut.

Usage

```

import { useEffect } from 'react';
import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect, useDispatch } from '@wordpress/data';
import { __ } from '@wordpress/i18n';

const ExampleComponent = () => {
    const { registerShortcut } = useDispatch( keyboardShortcutsStore );

    useEffect( () => {
        registerShortcut( {
            name: 'custom/my-custom-shortcut',
            category: 'my-category',
            description: __( 'My custom shortcut' ),
            keyCombination: {
                modifier: 'primary',
                character: 'j',
            },
        },
    );
}

```

```

        } );
    }, [ ] );

const shortcut = useSelect(
    ( select ) =>
        select( keyboardShortcutsStore ).getShortcutKeyCombination(
            'custom/my-custom-shortcut'
        ),
    [ ]
);

return shortcut ? (
    <p>{ __( 'Shortcut is registered.' ) }</p>
) : (
    <p>{ __( 'Shortcut is not registered.' ) }</p>
);
}

```

Parameters

- `config WPShortcutConfig`: Shortcut config.

Returns

- `Object`: action.

unregisterShortcut

Returns an action object used to unregister a keyboard shortcut.

Usage

```

import { useEffect } from 'react';
import { store as keyboardShortcutsStore } from '@wordpress/keyboard-shortcuts';
import { useSelect, useDispatch } from '@wordpress/data';
import { __ } from '@wordpress/i18n';

const ExampleComponent = () => {
    const { unregisterShortcut } = useDispatch( keyboardShortcutsStore );

    useEffect( () => {
        unregisterShortcut( 'core/edit-post/next-region' );
    }, [ ] );

    const shortcut = useSelect(
        ( select ) =>
            select( keyboardShortcutsStore ).getShortcutKeyCombination(
                'core/edit-post/next-region'
            ),
        [ ]
    );

    return shortcut ? (
        <p>{ __( 'Shortcut is not unregistered.' ) }</p>
    )
}

```

```
) : (
  <p>{ __( 'Shortcut is unregistered.' ) }</p>
);
};
```

Parameters

- `name` `string`: Shortcut name.

Returns

- `Object`: action.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: The Keyboard Shortcuts Data](#)

[Previous](#) [The Post Editor's Data](#) [Previous: The Post Editor's Data](#)

[Next](#) [Notices Data](#) [Next: Notices Data](#)

Notices Data

In this article

[Table of Contents](#)

- [Selectors](#)
 - [getNotices](#)
- [Actions](#)
 - [createErrorNotice](#)
 - [createInfoNotice](#)
 - [createNotice](#)
 - [createSuccessNotice](#)
 - [createWarningNotice](#)
 - [removeAllNotices](#)
 - [removeNotice](#)
 - [removeNotices](#)

[↑ Back to top](#)

Namespace: `core/notices`.

[Selectors](#)

getNotices

Returns all notices as an array, optionally for a given context. Defaults to the global context.

Usage

```
import { useSelect } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';

const ExampleComponent = () => {
    const notices = useSelect( ( select ) =>
        select( noticesStore ).getNotices()
    );
    return (
        <ul>
            { notices.map( ( notice ) => (
                <li key={ notice.ID }>{ notice.content }</li>
            ) ) }
        </ul>
    );
};
```

Parameters

- *state* Object: Notices state.
- *context* ?string: Optional grouping context.

Returns

- `WPNotice[]`: Array of notices.

Actions

createErrorNotice

Returns an action object used in signalling that an error notice is to be created. Refer to `createNotice` for options documentation.

Related

- `createNotice`

Usage

```
import { __ } from '@wordpress/i18n';
import { useDispatch } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    const { createErrorNotice } = useDispatch( noticesStore );
    return (
        <Button
            onClick={ () =>
```

```
        createErrorNotice( __( 'An error occurred!' ), {
            type: 'snackbar',
            explicitDismiss: true,
        } )
    }
>
{ __(
    'Generate an snackbar error notice with explicit dismiss b
) }
</Button>
);
}
};
```

Parameters

- `content string`: Notice message.
- `options [Object]`: Optional notice options.

Returns

- `Object`: Action object.

[createInfoNotice](#)

Returns an action object used in signalling that an info notice is to be created. Refer to `createNotice` for options documentation.

Related

- `createNotice`

Usage

```
import { __ } from '@wordpress/i18n';
import { useDispatch } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    const { createInfoNotice } = useDispatch( noticesStore );
    return (
        <Button
            onClick={ () =>
                createInfoNotice( __( 'Something happened!' ), {
                    isDismissible: false,
                } )
            }
        >
            { __( 'Generate a notice that cannot be dismissed.' ) }
        </Button>
    );
}
```

Parameters

- *content string*: Notice message.
- *options [Object]*: Optional notice options.

Returns

- *Object*: Action object.

[createNotice](#)

Returns an action object used in signalling that a notice is to be created.

Usage

```
import { __ } from '@wordpress/i18n';
import { useDispatch } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
  const { createNotice } = useDispatch( noticesStore );
  return (
    <Button
      onClick={ () => createNotice( 'success', __( 'Notice message' ) )
        { __( 'Generate a success notice!' ) }
      </Button>
  );
};
```

Parameters

- *status string|undefined*: Notice status (“info” if undefined is passed).
- *content string*: Notice message.
- *options [Object]*: Notice options.
 - *options.context [string]*: Context under which to group notice.
 - *options.id [string]*: Identifier for notice. Automatically assigned if not specified.
 - *options.isDismissible [boolean]*: Whether the notice can be dismissed by user.
 - *options.type [string]*: Type of notice, one of `default`, or `snackbar`.
 - *options.speak [boolean]*: Whether the notice content should be announced to screen readers.
 - *options.actions [Array<WPNoticeAction>]*: User actions to be presented with notice.
 - *options.icon [string]*: An icon displayed with the notice. Only used when type is set to `snackbar`.
 - *options.explicitDismiss [boolean]*: Whether the notice includes an explicit dismiss button and can't be dismissed by clicking the body of the notice. Only applies when type is set to `snackbar`.
 - *options.onDismiss [Function]*: Called when the notice is dismissed.

Returns

- *Object*: Action object.

[createSuccessNotice](#)

Returns an action object used in signalling that a success notice is to be created. Refer to `createNotice` for options documentation.

Related

- `createNotice`

Usage

```
import { __ } from '@wordpress/i18n';
import { useDispatch } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    const { createSuccessNotice } = useDispatch( noticesStore );
    return (
        <Button
            onClick={ () =>
                createSuccessNotice( __( 'Success!' ), {
                    type: 'snackbar',
                    icon: '🔥',
                } )
            }
        >
            { __( 'Generate a snackbar success notice!' ) }
        </Button>
    );
};
```

Parameters

- `content string`: Notice message.
- `options [Object]`: Optional notice options.

Returns

- `Object`: Action object.

[createWarningNotice](#)

Returns an action object used in signalling that a warning notice is to be created. Refer to `createNotice` for options documentation.

Related

- `createNotice`

Usage

```
import { __ } from '@wordpress/i18n';
import { useDispatch } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
```

```

import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    const { createWarningNotice, createInfoNotice } =
        useDispatch( noticesStore );
    return (
        <Button
            onClick={ () =>
                createWarningNotice( __( 'Warning!' ), {
                    onDismiss: () => {
                        createInfoNotice(
                            __( 'The warning has been dismissed!' )
                        );
                    },
                } )
            }
        >
            { __( 'Generates a warning notice with onDismiss callback' ) }
        </Button>
    );
};


```

Parameters

- `content` `string`: Notice message.
- `options` `[Object]`: Optional notice options.

Returns

- `Object`: Action object.

removeAllNotices

Removes all notices from a given context. Defaults to the default context.

Usage

```

import { __ } from '@wordpress/i18n';
import { useDispatch, useSelect } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
import { Button } from '@wordpress/components';

export const ExampleComponent = () => {
    const notices = useSelect( ( select ) =>
        select( noticesStore ).getNotices()
    );
    const { removeAllNotices } = useDispatch( noticesStore );
    return (
        <>
            <ul>
                { notices.map( ( notice ) => (
                    <li key={ notice.id }>{ notice.content }</li>
                ) ) }
            </ul>
    );
};

```

```

        <Button onClick={() => removeAllNotices()}>
            { __( 'Clear all notices', 'woo-gutenberg-products-block' )
        </Button>
        <Button onClick={() => removeAllNotices( 'snackbar' )}>
            { __(
                'Clear all snackbar notices',
                'woo-gutenberg-products-block'
            ) }
        </Button>
    </>
);
}
;
```

Parameters

- **noticeType** `string`: The context to remove all notices from.
- **context** `string`: The context to remove all notices from.

Returns

- `Object`: Action object.

[removeNotice](#)

Returns an action object used in signalling that a notice is to be removed.

Usage

```

import { __ } from '@wordpress/i18n';
import { useDispatch } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    const notices = useSelect( ( select ) =>
        select( noticesStore ).getNotices()
    );
    const { createWarningNotice, removeNotice } = useDispatch( noticesStor

    return (
        <>
            <Button
                onClick={ () =>
                    createWarningNotice( __( 'Warning!' ), {
                        isDismissible: false,
                    } )
                }
            >
                { __( 'Generate a notice' ) }
            </Button>
            { notices.length > 0 && (
                <Button onClick={ () => removeNotice( notices[ 0 ].id ) }>
                    { __( 'Remove the notice' ) }
                </Button>
            )}
        </>
    );
}
;
```

```
        )  }
    </>
);
};
```

Parameters

- *id* `string`: Notice unique identifier.
- *context* `[string]`: Optional context (grouping) in which the notice is intended to appear. Defaults to default context.

Returns

- `Object`: Action object.

[removeNotices](#)

Returns an action object used in signalling that several notices are to be removed.

Usage

```
import { __ } from '@wordpress/i18n';
import { useDispatch, useSelect } from '@wordpress/data';
import { store as noticesStore } from '@wordpress/notices';
import { Button } from '@wordpress/components';

const ExampleComponent = () => {
    const notices = useSelect( ( select ) =>
        select( noticesStore ).getNotices()
    );
    const { removeNotices } = useDispatch( noticesStore );
    return (
        <>
            <ul>
                { notices.map( ( notice ) => (
                    <li key={ notice.id }>{ notice.content }</li>
                ) ) }
            </ul>
            <Button
                onClick={ () =>
                    removeNotices( notices.map( ( { id } ) => id ) )
                }
            >
                { __( 'Clear all notices' ) }
            </Button>
        </>
    );
};
```

Parameters

- *ids* `string[]`: List of unique notice identifiers.
- *context* `[string]`: Optional context (grouping) in which the notices are intended to appear. Defaults to default context.

Returns

- `Object`: Action object.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Notices Data”](#)

[Previous The Keyboard Shortcuts Data](#) [Previous: The Keyboard Shortcuts Data](#)
[Next The NUX \(New User Experience\) Data](#) [Next: The NUX \(New User Experience\) Data](#)

The NUX (New User Experience) Data

In this article

[Table of Contents](#)

- [Selectors](#)
 - [areTipsEnabled](#)
 - [getAssociatedGuide](#)
 - [isTipVisible](#)
- [Actions](#)
 - [disableTips](#)
 - [dismissTip](#)
 - [enableTips](#)
 - [triggerGuide](#)

[↑ Back to top](#)

Namespace: `core/nux`.

[Selectors](#)

[areTipsEnabled](#)

Returns whether or not tips are globally enabled.

Parameters

- `state Object`: Global application state.

Returns

- `boolean`: Whether tips are globally enabled.

[getAssociatedGuide](#)

Returns an object describing the guide, if any, that the given tip is a part of.

Parameters

- `state Object`: Global application state.
- `tipId string`: The tip to query.

Returns

- `?NUXGuideInfo`: Information about the associated guide.

[isTipVisible](#)

Determines whether or not the given tip is showing. Tips are hidden if they are disabled, have been dismissed, or are not the current tip in any guide that they have been added to.

Parameters

- `state Object`: Global application state.
- `tipId string`: The tip to query.

Returns

- `boolean`: Whether or not the given tip is showing.

[Actions](#)

[disableTips](#)

Returns an action object that, when dispatched, prevents all tips from showing again.

Returns

- `Object`: Action object.

[dismissTip](#)

Returns an action object that, when dispatched, dismisses the given tip. A dismissed tip will not show again.

Parameters

- `id string`: The tip to dismiss.

Returns

- `Object`: Action object.

[enableTips](#)

Returns an action object that, when dispatched, makes all tips show again.

Returns

- `Object`: Action object.

[triggerGuide](#)

Returns an action object that, when dispatched, presents a guide that takes the user through a series of tips step by step.

Parameters

- `tipIds string []`: Which tips to show in the guide.

Returns

- `Object`: Action object.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: The NUX \(New User Experience\) Data”](#)

[Previous Notices Data](#) [Previous: Notices Data](#)

[Next Preferences](#) [Next: Preferences](#)

Preferences

In this article

Table of Contents

- [Selectors](#)
 - [get](#)
- [Actions](#)
 - [set](#)
 - [setDefaults](#)
 - [setPersistenceLayer](#)
 - [toggle](#)

[↑ Back to top](#)

Namespace: core/preferences.

Selectors

get

Returns a boolean indicating whether a prefer is active for a particular scope.

Parameters

- *state Object*: The store state.
- *scope string*: The scope of the feature (e.g. core/edit-post).
- *name string*: The name of the feature.

Returns

- *: Is the feature enabled?

Actions

set

Returns an action object used in signalling that a preference should be set to a value

Parameters

- *scope string*: The preference scope (e.g. core/edit-post).
- *name string*: The preference name.
- *value **: The value to set.

Returns

- *Object*: Action object.

setDefaults

Returns an action object used in signalling that preference defaults should be set.

Parameters

- *scope string*: The preference scope (e.g. core/edit-post).
- *defaults Object<string, *>*: A key/value map of preference names to values.

Returns

- *Object*: Action object.

setPersistenceLayer

Sets the persistence layer.

When a persistence layer is set, the preferences store will:

- call `get` immediately and update the store state to the value returned.
- call `set` with all preferences whenever a preference changes value.

`setPersistenceLayer` should ideally be dispatched at the start of an application's lifecycle, before any other actions have been dispatched to the preferences store.

Parameters

- `persistenceLayer` `WPPreferencesPersistenceLayer`: The persistence layer.

Returns

- `Object`: Action object.

[toggle](#)

Returns an action object used in signalling that a preference should be toggled.

Parameters

- `scope string`: The preference scope (e.g. core/edit-post).
- `name string`: The preference name.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Preferences”](#)

[Previous The NUX \(New User Experience\) Data](#) [Previous: The NUX \(New User Experience\) Data](#)

[Next Reusable blocks](#) [Next: Reusable blocks](#)

Reusable blocks

In this article

[Table of Contents](#)

- [Selectors](#)
- [Actions](#)

[↑ Back to top](#)

Namespace: `core/reusable-blocks`.

This package is still experimental. “Experimental” means this is an early implementation subject to drastic and breaking changes.

Selectors

Nothing to document.

Actions

Nothing to document.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Reusable blocks](#)”

[Previous Preferences](#) [Previous: Preferences](#)

[Next Rich Text](#) [Next: Rich Text](#)

Rich Text

In this article

Table of Contents

- [Selectors](#)
 - [getFormatType](#)
 - [getFormatTypeForBareElement](#)
 - [getFormatTypeForClassName](#)
 - [getFormatTypes](#)
- [Actions](#)

[↑ Back to top](#)

Namespace: `core/rich-text`.

Selectors

[getFormatType](#)

Returns a format type by name.

Usage

```
import { __, sprintf } from '@wordpress/i18n';
import { store as richTextStore } from '@wordpress/rich-text';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const { getFormatType } = useSelect(
        ( select ) => select( richTextStore ),
        []
    );

    const boldFormat = getFormatType( 'core/bold' );

    return boldFormat ? (
        <ul>
            { Object.entries( boldFormat )?.map( ( [ key, value ] ) => (
                <li>
                    { key } : { value }
                </li>
            ) ) }
        </ul>
    ) : (
        __( 'Not Found' )
    );
};
```

Parameters

- *state* Object: Data state.
- *name* string: Format type name.

Returns

- Object?: Format type.

[getFormatTypeForBareElement](#)

Gets the format type, if any, that can handle a bare element (without a data-format-type attribute), given the tag name of this element.

Usage

```
import { __, sprintf } from '@wordpress/i18n';
import { store as richTextStore } from '@wordpress/rich-text';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const { getFormatTypeForBareElement } = useSelect(
        ( select ) => select( richTextStore ),
```

```
        []
);
const format = getFormatTypeForBareElement( 'strong' );
return format && <p>{ sprintf( __( 'Format name: %s' ), format.name )
};
```

Parameters

- *state Object*: Data state.
- *bareElementTagName string*: The tag name of the element to find a format type for.

Returns

- ?Object: Format type.

[getFormatTypeForClassName](#)

Gets the format type, if any, that can handle an element, given its classes.

Usage

```
import { __, sprintf } from '@wordpress/i18n';
import { store as richTextStore } from '@wordpress/rich-text';
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const { getFormatTypeForClassName } = useSelect(
        ( select ) => select( richTextStore ),
        []
    );

    const format = getFormatTypeForClassName( 'has-inline-color' );

    return format && <p>{ sprintf( __( 'Format name: %s' ), format.name )
};
```

Parameters

- *state Object*: Data state.
- *elementClassName string*: The classes of the element to find a format type for.

Returns

- ?Object: Format type.

[getFormatTypes](#)

Returns all the available format types.

Usage

```
import { __, sprintf } from '@wordpress/i18n';
import { store as richTextStore } from '@wordpress/rich-text';
```

```
import { useSelect } from '@wordpress/data';

const ExampleComponent = () => {
    const { getFormatTypes } = useSelect(
        ( select ) => select( richTextStore ),
        []
    );

    const availableFormats = getFormatTypes();

    return availableFormats ? (
        <ul>
            { availableFormats?.map( ( format ) => (
                <li>{ format.name }</li>
            ) ) }
        </ul>
    ) : (
        __( 'No Formats available' )
    );
};
```

Parameters

- *state* Object: Data state.

Returns

- Array: Format types.

[Actions](#)

Nothing to document.

First published

July 26, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Rich Text”](#)

[Previous Reusable blocks](#) [Previous: Reusable blocks](#)

[Next The Viewport Data](#) [Next: The Viewport Data](#)

The Viewport Data

In this article

Table of Contents

- [Selectors](#)
 - [isViewportMatch](#)
- [Actions](#)

[↑ Back to top](#)

Namespace: core/viewport.

[Selectors](#)

[isViewportMatch](#)

Returns true if the viewport matches the given query, or false otherwise.

Usage

```
import { store as viewportStore } from '@wordpress/viewport';
import { useSelect } from '@wordpress/data';
import { __ } from '@wordpress/i18n';
const ExampleComponent = () => {
    const isMobile = useSelect(
        ( select ) => select( viewportStore ).isViewportMatch( '< small' )
        []
    );
    return isMobile ? (
        <div>{ __( 'Mobile' ) }</div>
    ) : (
        <div>{ __( 'Not Mobile' ) }</div>
    );
};
```

Parameters

- *state Object*: Viewport state object.
- *query string*: Query string. Includes operator and breakpoint name, space separated.
Operator defaults to \geq .

Returns

- *boolean*: Whether viewport matches query.

Actions

The actions in this package shouldn't be used directly.

Nothing to document.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: The Viewport Data”](#)

[Previous Rich Text](#) [Previous: Rich Text](#)

[Next Explanations](#) [Next: Explanations](#)

Explanations

[↑ Back to top](#)

Architecture

- [Key Concepts](#)
- [Data Format And Data Flow](#)
- [Modularity and WordPress Packages](#).
- [Block Editor Performance](#).
- What are the decision decisions behind the Data Module?
- [Why is Puppeteer the tool of choice for end-to-end tests?](#)
- [What's the difference between the different editor packages? What's the purpose of each package?](#)
- [Template and template parts flows](#)

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Explanations”](#)

[Previous The Viewport Data](#) [Previous: The Viewport Data](#)

[Next Architecture](#) [Next: Architecture](#)

Architecture

In this article

[Table of Contents](#)

- [Editor](#)
- [Gutenberg repository](#)

[↑ Back to top](#)

Let's look at the big picture and the architectural and UX principles of the block editor and the Gutenberg repository.

[Editor](#)

- [Key concepts](#).
- [Data format and data flow](#).
- [Entities and undo/redo](#).
- [Site editing templates](#).
- [Styles in the editor](#).
- [Performance](#).

[Gutenberg repository](#)

- [Modularity and WordPress Packages](#).
- [Understand the repository folder structure](#).
- **Outdated!** [Why is Puppeteer the tool of choice for end-to-end tests?](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Architecture”](#)

[Previous Explanations](#) [Previous: Explanations](#)
[Next Key Concepts](#) [Next: Key Concepts](#)

Key Concepts

In this article

[Table of Contents](#)

- [Blocks](#)
 - [Composability](#)
 - [Data and attributes](#)
 - [Block transforms](#)
 - [Block variations](#)
- [Reusable blocks](#)
- [Patterns](#)
- [Templates](#)
- [Styles](#)

[↑ Back to top](#)

[Blocks](#)

Blocks are an abstract unit for structuring and interacting with content. When composed together they create the content for a webpage. Everything from a paragraph, to a video, to the site title is represented as a block.

Blocks come in many different forms but also provide a consistent interface. They can be inserted, moved, reordered, copied, duplicated, transformed, deleted, dragged, and combined. Blocks can also be reused, allowing them to be shared across posts and post types and/or used multiple times in the same post. If it helps, you can think of blocks as a more graceful shortcode, with rich formatting tools for users to compose content.

The settings and content of a block can be customized in three main places: the block canvas, the block toolbar, and the block inspector.

[Composability](#)

Blocks are meant to be combined in different ways. Blocks are hierarchical in that a block can be nested within another block. Nested blocks and its container are also called *children* and *parent* respectively. For example, a *Columns* block can be the parent block to multiple child blocks in each of its columns. The API that governs child block usage is named `InnerBlocks`.

[Data and attributes](#)

Blocks understand content as attributes and are serializable to HTML. To this point, there is a new Block Grammar. Distilled, the block grammar is an HTML comment, either a self-closing tag or with a beginning tag and ending tag. In the main tag, depending on the block type and user customizations, there can be a JSON object. This raw form of the block is referred to as serialized.

```
<!-- wp:paragraph {"key": "value"} -->
<p>Welcome to the world of blocks.</p>
<!-- /wp:paragraph -->
```

Blocks can be static or dynamic. Static blocks contain rendered content and an object of Attributes used to re-render based on changes. Dynamic blocks require server-side data and rendering while the post content is being generated (rendering).

Each block contains Attributes or configuration settings, which can be sourced from raw HTML in the content via meta or other customizable origins.

More on [Data format and data flow](#).

Block transforms

Blocks have the ability to be transformed into other block types. This allows basic operations like converting a paragraph into a heading, but also more intricate ones like multiple images becoming a gallery. Block transforms work for single blocks and for multi-block selections. Internal block variations are also possible transformation targets.

Block variations

Given a block type, a block variation is a predefined set of its initial attributes. This API allows creating a single block from which multiple configurations are possible. Variations provide different possible interfaces, including showing up as entirely new blocks in the library, or as presets when inserting a new block. Read [the API documentation](#) for more details.

More on blocks

- [Block API](#)
- [Tutorial: Building A Custom Block](#)

Reusable blocks

A reusable blocks is **an instance** of a block (or multiple blocks) that can be inserted and edited in multiples places, remaining in sync everywhere. If a reusable block is edited in one place, those changes are reflected across all posts and pages that block is used. Examples of reusable blocks include a block consisting of a heading whose content and a custom color that would appear on multiple pages of the site and sidebar widgets that would appear on every page.

Any edits to a reusable block will appear on every other use of that block, saving time from having to make the same edit on different posts.

Internally, reusable blocks are stored as a hidden post type (`wp_block`) and are dynamic blocks that “ref” or reference the `post_id` and return the `post_content` for that block.

Patterns

A [block pattern](#) is a group of blocks that have been combined together creating a design pattern. These design patterns provide a starting point for building more advanced pages and layouts quickly, instead of inserting individual blocks. A block pattern can be as small as a single block or as large as a full page of content. Unlike reusable blocks, once a pattern is inserted it doesn’t remain in sync with the original content as the blocks contained are meant to be edited and

customized by the user. Underneath the surface, patterns are just regular blocks composed together. Themes can register patterns to offer users quick starting points with a design language familiar to that theme's aesthetics.

Templates

While the post editor concentrates on the content of a post, the [template](#) editor allows declaring and editing an entire site using blocks, from header to footer. Templates are broken down between templates (that describe a full page) and template parts (that describe reusable areas within a template, including semantic areas like header, sidebar, and footer).

These templates and template parts can be composed together and registered by a theme. They are also entirely editable by users using the block editor; a collection of blocks that interact with different properties and settings of the site (like the site title, description, logo, navigation, etc) are especially useful when editing templates and template parts. Customized templates are saved in a `wp_template` post type. Block templates include both static pages and dynamic ones, like archives, singular, home, 404, etc.

Note: custom post types can also be initialized with a starting `post_content` template that should not be confused with the theme template system described above.

More on [Site editing templates](#).

Styles

Styles, formerly known as Global Styles and as such referenced in the code, is both an interface that users access through the editor and a configuration system done through [a theme.json file](#). This file absorbs most of the configuration aspects usually scattered through various `add_theme_support` calls to simplify communicating with the editor. It thus aims to improve declaring what settings should be enabled, what specific tools a theme offers (like a custom color palette), the available design tools present, and an infrastructure that allows to coordinate the styles coming from WordPress, the active theme, and the user.

Learn more about [Global Styles](#).

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Key Concepts”](#)

[Previous Architecture](#) [Previous: Architecture](#)

[Next Data Flow and Data Format](#) [Next: Data Flow and Data Format](#)

Data Flow and Data Format

In this article

[Table of Contents](#)

- [The format](#)
 - [The block object](#)
- [Serialization and parsing](#)
 - [Delimiters and parsing expression grammar](#)
 - [The anatomy of a serialized block](#)
- [The data lifecycle](#)

[↑ Back to top](#)

[The format](#)

A block editor post is the proper block-aware representation of a post: a collection of semantically consistent descriptions of what each block is and what its essential data is. This representation only ever exists in memory. It is the [chase](#) in the typesetter's workshop, ever-shifting as [sorts](#) are attached and repositioned.

A block editor post is not the artifact it produces, namely the `post_content`. The latter is the printed page, optimized for the reader but retaining its invisible markings for later editing.

The input and output of the block editor is a tree of block objects with the current format:

```
const value = [ block1, block2, block3 ];
```

[The block object](#)

Each block object has an `id`, a set of attributes and potentially a list of child blocks.

```
const block = {  
  clientId, // unique string identifier.  
  type, // The block type (paragraph, image...)  
  attributes, // (key, value) set of attributes representing the direct  
  innerBlocks, // An array of child blocks or inner blocks.  
};
```

Note the attributes keys and types, the allowed inner blocks are defined by the block type. For example, the core quote block has a `cite` string attribute representing the cite content while a heading block has a numeric `level` attribute, representing the level of the heading (1 to 6).

During the lifecycle of the block in the editor, the block object can receive extra metadata:

- `isValid`: A boolean representing whether the block is valid or not;
- `originalContent`: The original HTML serialization of the block.

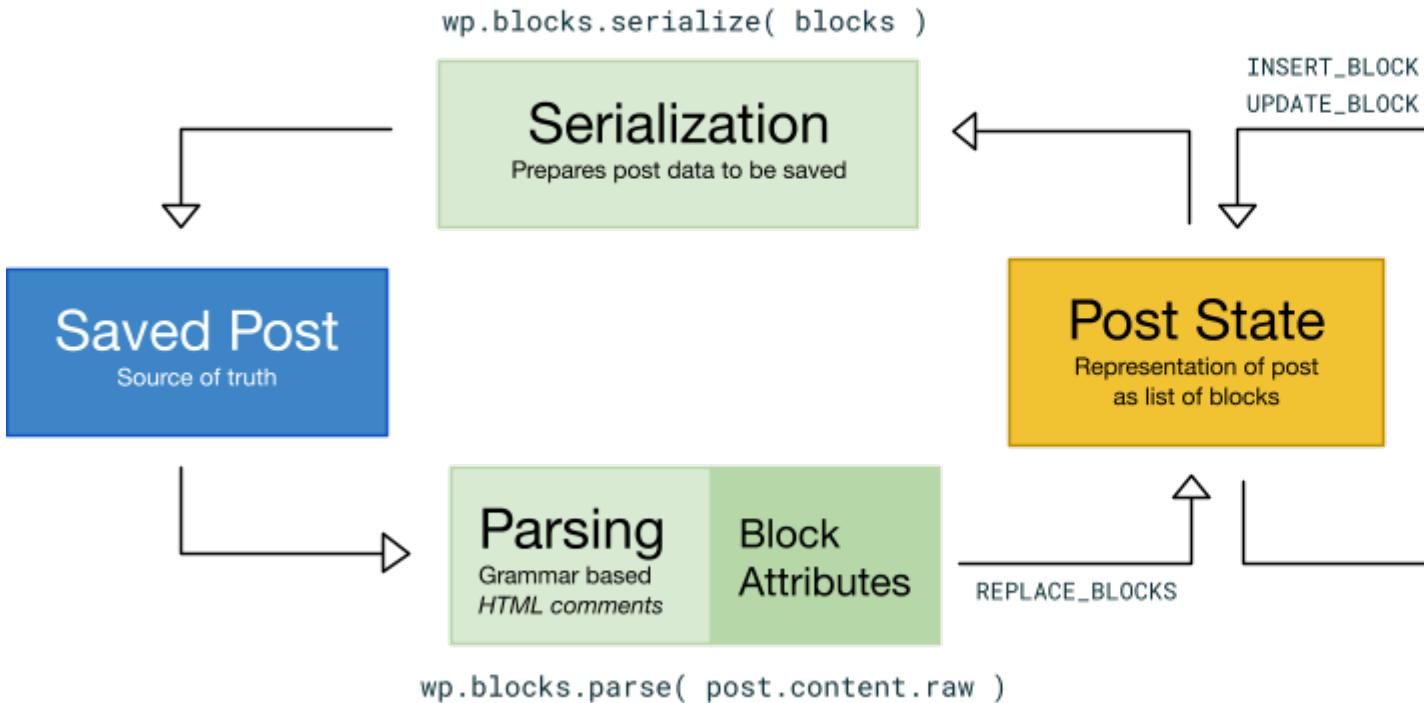
Examples

```
// A simple paragraph block.
const paragraphBlock1 = {
  clientId: '51828be1-5f0d-4a6b-8099-f4c6f897e0a3',
  type: 'core/paragraph',
  attributes: {
    content: 'This is the <strong>content</strong> of the paragraph b
    dropCap: true,
  },
};

// A separator block.
const separatorBlock = {
  clientId: '51828be1-5f0d-4a6b-8099-f4c6f897e0a4',
  type: 'core/separator',
  attributes: {},
};

// A columns block with a paragraph block on each column.
const columnsBlock = {
  clientId: '51828be1-5f0d-4a6b-8099-f4c6f897e0a7',
  type: 'core/columns',
  attributes: {},
  innerBlocks: [
    {
      clientId: '51828be1-5f0d-4a6b-8099-f4c6f897e0a5',
      type: 'core/column',
      attributes: {},
      innerBlocks: [ paragraphBlock1 ],
    },
    {
      clientId: '51828be1-5f0d-4a6b-8099-f4c6f897e0a6',
      type: 'core/column',
      attributes: {},
      innerBlocks: [ paragraphBlock2 ],
    },
  ],
};
```

Serialization and parsing



This data model, however, is something that lives in memory while editing a post. It's not visible to the page viewer when rendered, just like a printed page has no trace of the structure of the letters that produced it in the press.

Since the whole WordPress ecosystem has an expectation for receiving HTML when rendering or editing a post, the block editor transforms its data into something that can be saved in `post_content` through serialization. This assures that there's a single source of truth for the content, and that this source remains readable and compatible with all the tools that interact with WordPress content at the present. Were we to store the object tree separately, we would face the risk of `post_content` and the tree getting out of sync and the problem of data duplication in both places.

Thus, the serialization process converts the block tree into HTML using HTML comments as explicit block delimiters—which can contain the attributes in non-HTML form. This is the act of printing invisible marks on the printed page that leave a trace of the original structured intention.

This is one end of the process. The other is how to recreate the collection of blocks whenever a post is to be edited again. A formal grammar defines how the serialized representation of a block editor post should be loaded, just as some basic rules define how to turn the tree into an HTML-like string. The block editor's posts aren't designed to be edited by hand; they aren't designed to be edited as HTML documents because the block editor posts aren't HTML in essence.

They just happen, incidentally, to be stored inside of `post_content` in a way in which they require no transformation in order to be viewable by any legacy system. It's true that loading the stored HTML into a browser without the corresponding machinery might degrade the experience,

and if it included dynamic blocks of content, the dynamic elements may not load, server-generated content may not appear, and interactive content may remain static. However, it at least protects against not being able to view block editor posts on themes and installations that are blocks-unaware, and it provides the most accessible way to the content. In other words, the post remains mostly intact even if the saved HTML is rendered as is.

Delimiters and parsing expression grammar

We chose instead to try to find a way to keep the formality, explicitness, and unambiguity in the existing HTML syntax. Within the HTML there were a number of options.

Of these options, a novel approach was suggested: by storing data in HTML comments, we would know that we wouldn't break the rest of the HTML in the document, that browsers should ignore it, and that we could simplify our approach to parsing the document.

Unique to HTML comments is the fact that they cannot legitimately exist in ambiguous places, such as inside of HTML attributes like ``. Comments are also quite permissive. Whereas HTML attributes are complicated to parse properly, comments are quite easily described by a leading `<!--` followed by anything except `-->` until the first `-->`. This simplicity and permissiveness means that the parser can be implemented in several ways without needing to understand HTML properly, and we have the liberty to use more convenient syntax inside of the comment—we only need to escape double-hyphen sequences. We take advantage of this in how we store block attributes: as JSON literals inside the comment.

After running this through the parser, we're left with a simple object we can manipulate idiomatically, and we don't have to worry about escaping or unescaping the data. It's handled for us through the serialization process. Because the comments are so different from other HTML tags and because we can perform a first-pass to extract the top-level blocks, we don't actually depend on having fully valid HTML!

This has dramatic implications for how simple and performant we can make our parser. These explicit boundaries also protect damage in a single block from bleeding into other blocks or tarnishing the entire document. It also allows the system to identify unrecognized blocks before rendering them.

N.B.: The defining aspects of blocks are their semantics and the isolation mechanism they provide: in other words, their identity. On the other hand, where their data is stored is a more liberal aspect. Blocks support more than just static local data (via JSON literals inside the HTML comment or within the block's HTML), and more mechanisms (e.g., global blocks or otherwise resorting to storage in complementary `WP_Post` objects) are expected. See [attributes](#) for details.

The anatomy of a serialized block

When blocks are saved to the content after the editing session, its attributes—depending on the nature of the block—are serialized to these explicit comment delimiters.

```
<!-- wp:image -->
<figure class="wp-block-image"></figure>
<!-- /wp:image -->
```

A purely dynamic block that is to be server-rendered before display could look like this:

```
<!-- wp:latest-posts {"postsToShow":4,"displayPostDate":true} /-->
```

The data lifecycle

In summary, the block editor workflow parses the saved document to an in-memory tree of blocks, using token delimiters to help. During editing, all manipulations happen within the block tree. The process ends by serializing the blocks back to the `post_content`.

The workflow process relies on a serialization/parser pair to persist posts. Hypothetically, the post data structure could be stored using a plugin or retrieved from a remote JSON file to be converted to the block tree.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Data Flow and Data Format”](#)

[Previous Key Concepts](#) [Previous: Key Concepts](#)

[Next Entities and Undo/Redo.](#) [Next: Entities and Undo/Redo.](#)

Entities and Undo/Redo.

In this article

Table of Contents

- [Editing entities](#)
- [Undo/Redo](#)
 - [Cached changes](#)

[↑ Back to top](#)

The WordPress editors, whether it's the post or site editor, manipulate what we call entity records. These are objects that represent a post, a page, a user, a term, a template, etc. They are the data that is stored in the database and that is manipulated by the editor. Each editor can fetch, edit and save multiple entity records at the same time.

For instance, when opening a page in the site editor:

- you can edit properties of the page itself (title, content...)
- you can edit properties of the template of the page (content of the template, design...)
- you can edit properties of template parts (header, footer) used with the template.

The editor keeps track of all these modifications and orchestrates the saving of all these modified records. This happens within the `@wordpress/core-data` package.

Editing entities

To be able to edit an entity, you need to first fetch it and load it into the `core-data` store. For example, the following code loads the post with ID 1 into the store. (The entity is the post, the post 1 is the entity record).

```
wp.data.select( 'core' ).getEntityRecord( 'postType', 'post', 1 );
```

Once the entity is loaded, you can edit it. For example, the following code sets the title of the post to “Hello World”. For each fetched entity record, the `core-data` store keeps track of:

- the “persisted” record: The last state of the record as it was fetched from the backend.
- A list of “edits”: Unsaved local modifications for one or several properties of the record.

The package also exposes a set of actions to manipulate the fetched entity records.

To edit an entity record, you can call `editEntityRecord`, which takes the entity type, the entity ID and the new entity record as parameters. The following example sets the title of the post with ID 1 to “Hello World”.

```
wp.data.dispatch( 'core' ).editEntityRecord( 'postType', 'post', 1, { title: 'Hello World' } );
```

Once you have edited an entity record, you can save it. The following code saves the post with ID 1.

```
wp.data.dispatch( 'core' ).saveEditedEntityRecord( 'postType', 'post', 1 )
```

Undo/Redo

Since the WordPress editors allow multiple entity records to be edited at the same time, the `core-data` package keeps track of all the entity records that have been fetched and edited in a common undo/redo stack. Each step in the undo/redo stack contains a list of “edits” that should be undone or redone at the same time when calling the `undo` or `redo` action.

And to be able to perform both undo and redo operations properly, each modification in the list of edits contains the following information:

- Entity kind and name: Each entity in core-data is identified by the pair *(kind, name)*. This corresponds to the identifier of the modified entity.
- Entity Record ID: The ID of the modified record.
- Property: The name of the modified property.
- From: The previous value of the property (needed to apply the undo operation).
- To: The new value of the property (needed to apply the redo operation).

For example, let’s say a user edits the title of a post, followed by a modification to the post slug, and then a modification of the title of a reusable block used with the post. The following information is stored in the undo/redo stack:

- [{ kind: 'postType', name: 'post', id: 1, property: 'title', from: '', to: 'Hello World' }]
- [{ kind: 'postType', name: 'post', id: 1, property: 'slug', from: 'Previous slug', to: 'This is the slug of the hello world post' }]

- [{ kind: 'postType', name: 'wp_block', id: 2, property: 'title', from: 'Reusable Block', to: 'Awesome Reusable Block' }]

The store also keep tracks of a “pointer” to the current “undo/redo” step. By default, the pointer always points to the last item in the stack. This pointer is updated when the user performs an undo or redo operation.

[Cached changes](#)

The undo/redo core behavior also supports what we call “cached modifications”. These are modifications that are not stored in the undo/redo stack right away. For instance, when a user starts typing in a text field, the value of the field is modified in the store, but this modification is not stored in the undo/redo stack until after the user moves to the next word or after a few milliseconds. This is done to avoid creating a new undo/redo step for each character typed by the user.

Cached changes are kept outside the undo/redo stack in what is called a “cache” of modifications and these modifications are only stored in the undo/redo stack when we explicitly call `__unstableCreateUndoLevel` or when the next modification is not a cached one.

By default all calls to `editEntityRecord` are considered “non-cached” unless the `isCached` option is passed as true. Example:

```
wp.data.dispatch( 'core' ).editEntityRecord( 'postType', 'post', 1, { title: 'My Post' } )
```

First published

May 29, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Entities and Undo/Redo.”](#)

[Previous Data Flow and Data Format](#) [Previous: Data Flow and Data Format](#)
[Next Modularity](#) [Next: Modularity](#)

Modularity

In this article

Table of Contents

- [Why?](#)
- [Types of packages](#)
 - [Production packages](#)
 - [Development packages](#)

- [Editor packages](#)
 - [What's the difference between the different editor packages? What's the purpose of each package?](#)
- [Going further](#)

[↑ Back to top](#)

The WordPress block editor is based around the idea that you can combine independent blocks together to write your post or build your page. Blocks can also use and interact with each other. This makes it very modular and flexible.

But the Block Editor does not embrace modularity for its behavior and output only. The Gutenberg repository is also built from the ground up as several reusable and independent modules or packages, that, combined together, lead to the application and interface we all know. These modules are known as [WordPress packages](#) and are published and updated regularly on the npm package repository.

These packages are used to power the Block Editor, but they can be used to power any page in the WordPress Admin or outside.

[Why?](#)

Using a modular architecture has several benefits for all the actors involved:

- Each package is an independent unit and has a well defined public API that is used to interact with other packages and third-party code. This makes it easier for **Core Contributors** to reason about the codebase. They can focus on a single package at a time, understand it and make updates while knowing exactly how these changes could impact all the other parts relying on the given package.
- A module approach is also beneficial to the **end-user**. It allows to selectively load scripts on different WordPress Admin pages while keeping the bundle size contained. For instance, if we use the components package to power our plugin's settings page, there's no need to load the block-editor package on that page.
- This architecture also allows **third-party developers** to reuse these packages inside and outside the WordPress context by using these packages as npm or WordPress script dependencies.

[Types of packages](#)

Almost everything in the Gutenberg repository is built into a package. We can split these packages into two different types:

[Production packages](#)

These are the packages that ship in WordPress itself as JavaScript scripts. These constitute the actual production code that runs on your browsers. As an example, there's a `components` package serving as a reusable set of React components used to prototype and build interfaces quickly. There's also an `api-fetch` package that can be used to call WordPress Rest APIs.

Third-party developers can use these production packages in two different ways:

- If you're building a JavaScript application, website, page that runs outside of the context of WordPress, you can consume these packages like any other JavaScript package in the npm registry.

```
npm install @wordpress/components

import { Button } from '@wordpress/components';

function MyApp() {
    return <Button>Nice looking button</Button>;
}
```

- If you're building a plugin that runs on WordPress, you'd probably prefer consuming the package that ships with WordPress itself. This allows multiple plugins to reuse the same packages and avoid code duplication. In WordPress, these packages are available as WordPress scripts with a handle following this format `wp-package-name` (e.g. `wp-components`). Once you add the script to your own WordPress plugin scripts dependencies, the package will be available on the `wp` global variable.

```
// myplugin.php
// Example of script registration depending on the "components" and "el"
wp_register_script( 'myscript', 'path to myscript.js', array ('wp-components'

// Using the package in your scripts
const { Button } = wp.components;

function MyApp() {
    return <Button>Nice looking button</Button>;
}
```

Script dependencies definition can be a tedious task for developers. Mistakes and oversight can happen easily. If you want to learn how you can automate this task. Check the [@wordpress/scripts](#) and [@wordpress/dependency-extraction-webpack-plugin](#) documentation.

Packages with stylesheets

Some production packages provide stylesheets to function properly.

- If you're using the package as an npm dependency, the stylesheets will be available on the `build-style` folder of the package. Make sure to load this style file on your application.
- If you're working in the context of WordPress, you'll have to enqueue these stylesheets or add them to your stylesheets dependencies. The stylesheet handles are the same as the script handles.

In the context of existing WordPress pages, if you omit to define the scripts or styles dependencies properly, your plugin might still work properly if these scripts and styles are already loaded there by WordPress or by other plugins, but it's highly recommended to define all your dependencies exhaustively if you want to avoid potential breakage in future versions.

Packages with data stores

Some WordPress production packages define data stores to handle their state. These stores can also be used by third-party plugins and themes to retrieve data and to manipulate it. The name of these data stores is also normalized following this format `core/package-name` (E.g. the `@wordpress/block-editor` package defines and uses the `core/block-editor` data store).

If you're using one of these stores to access and manipulate WordPress data in your plugins, don't forget to add the corresponding WordPress script to your own script dependencies for your plugin to work properly. (For instance, if you're retrieving data from the `core/block-editor` store, you should add the `wp-block-editor` package to your script dependencies like shown above).

Development packages

These are packages used in development mode to help developers with daily tasks to develop, build and ship JavaScript applications, WordPress plugins and themes. They include tools for linting your codebase, building it, testing it...

Editor packages

WP Admin

post.php

Initialize
the post editor

①

retrieve
edited
post

②

save cha

⑨

What's the difference between the different editor packages? What's the purpose of each package?

It's often surprising to new contributors to discover that the post editor is constructed as a layered abstraction of three separate packages `@wordpress/edit-post`, `@wordpress/editor`, and `@wordpress/block-editor`.

The above [Why?](#) section should provide some context for how individual packages aim to satisfy specific requirements. That applies to these packages as well:

- `@wordpress/block-editor` provides components for implementing a block editor, operating on a primitive value of an array of block objects. It makes no assumptions for how this value is saved, and has no awareness (or requirement) of a WordPress site.
- `@wordpress/editor` is the enhanced version of the block editor for WordPress posts. It utilizes components from the `@wordpress/block-editor` package. Having an awareness of the concept of a WordPress post, it associates the loading and saving mechanism of the value representing blocks to a post and its content. It also provides various components relevant for working with a post object in the context of an editor (e.g., a post title input component). This package can support editing posts of any post type and does not assume that rendering happens in any particular WordPress screen or layout arrangement.
- `@wordpress/edit-post` is the implementation of the “New Post” (“Edit Post”) screen in the WordPress admin. It is responsible for the layout of the various components provided by `@wordpress/editor` and `@wordpress/block-editor`, with full awareness of how it is presented in the specific screen in the WordPress administrative dashboard.

Structured this way, these packages can be used in a variety of combinations outside the use-case of the “New Post” screen:

- A `@wordpress/edit-site` or `@wordpress/edit-widgets` package can serve as similar implementations of a “Site Editor” or “Widgets Editor”, in much the same way as `@wordpress/edit-post`.
- `@wordpress/editor` could be used in the implementation of the “Reusable Block” block, since it is essentially a nested block editor associated with the post type `wp_block`.
- `@wordpress/block-editor` could be used independently from WordPress, or with a completely different save mechanism. For example, it could be used for a comments editor for posts of a site.

Going further

- [Package Reference](#)

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Modularity](#)

Performance

In this article

Table of Contents

- [Metrics](#)
- [Key performance decisions and solutions](#)
- [The performance benchmark job](#)
- [Going further](#)

[↑ Back to top](#)

Performance is a key feature for editor applications and the Block editor is not an exception.

Metrics

To ensure the block editor stays performant across releases and development, we monitor some key metrics using [performance benchmark job](#).

- **Loading Time:** The time it takes to load an editor page. This includes time the server takes to respond, times to first paint, first contentful paint, DOM content load complete, load complete and first block render.
- **Typing Time:** The time it takes for the browser to respond while typing on the editor.
- **Block Selection Time:** The time it takes for the browser to respond after a user selects block. (Inserting a block is also equivalent to selecting a block. Monitoring the selection is sufficient to cover both metrics).

Key performance decisions and solutions

Data Module Async Mode

The Data Module of the WordPress Packages and the Block Editor is based on Redux. It means the state is kept globally and whenever a change happens, the components (UI) relying on that state may update.

As the number of rendered components grows (for example on long posts), the performance suffers because of the global state acting as an event dispatcher to all components. This is a common pitfall in Redux applications and the issue is solved on Gutenberg using the Data Modules Async Mode.

The Async mode is the idea that you can decide whether to refresh/rerender a part of the React component tree synchronously or asynchronously.

Rendering asynchronously in this context means that if a change is triggered in the global state, the subscribers (components) are not called synchronously, instead, we wait for the browser to be idle and perform the updates to React Tree.

Based on the idea that **when editing a given block, it is very rare that an update to that block affects other parts of the content**, the block editor canvas only renders the selected block in synchronous mode, all the remaining blocks are rendered asynchronously. This ensures that the editor stays responsive as the post grows.

The performance benchmark job

A tool to compare performance across multiple branches/tags/commits is provided. You can run it locally like so: `./bin/plugin/cli.js perf [branches]`, example:

```
./bin/plugin/cli.js perf trunk v8.1.0 v8.0.0
```

To get the most accurate results, it's important to use the exact same version of the tests and environment (theme...) when running the tests, the only thing that needs to be different between the branches is the Gutenberg plugin version (or branch used to build the plugin).

To achieve that the command first prepares the following folder structure:

```
└── tests/packages/e2e-tests/specs/performance/*
    The actual performance tests to run

└── tests/test/emptytheme
    The theme used for the tests environment. (site editor)

└── envs/branch1/.wp-env.json
    The wp-env config file for branch1 (similar to all other branches except)
└── envs/branch1/plugin
    A built clone of the Gutenberg plugin for branch1 (git checkout branch1)

└── envs/branchX
    The structure of perf-envs/branch1 is duplicated for all other branches
```

Once the directory above is in place, the performance command loops over the performance test suites (post editor and site editor) and does the following:

1. Start the environment for `branch1`
2. Run the performance test for the current suite
3. Stop the environment for `branch1`
4. Repeat the first 3 steps for all other branches
5. Repeat the previous 4 steps 3 times.
6. Compute medians for all the performance metrics of the current suite.

Once all the test suites are executed, a summary report is printed.

Going further

- [Journey towards a performant editor](#)

First published

March 9, 2021

Last updated

January 29, 2024

[Edit article](#)

[Improve it on GitHub: Performance”](#)

[Previous Modularity](#) [Previous: Modularity](#)

[Next Automated Testing](#) [Next: Automated Testing](#)

Automated Testing

[↑ Back to top](#)

Why is Puppeteer the tool of choice for end-to-end tests?

There exists a rich ecosystem of tooling available for web-based end-to-end automated testing. Thus, it's a common question: “Why does Gutenberg use [Puppeteer](#) instead of ([Cypress](#), [Selenium](#), [Playwright](#), etc)?”. Given some historical unreliability of the build results associated with end-to-end tests, it's especially natural to weigh this question in considering whether our tools are providing more value than the effort required in maintaining them. While we should always be comfortable in reevaluating earlier decisions, there were and continue to be many reasons that Puppeteer is the best compromise of the options available for end-to-end testing.

These include:

- **Interoperability with existing testing framework.** Puppeteer is “just” a tool for controlling a Chrome browser, and makes no assumptions about how it's integrated into a testing environment. While this requires some additional effort in ensuring the test environment is available, it also allows for cohesion in how it integrates with an existing setup. Gutenberg is able to consistently use Jest for both unit testing and end-to-end testing. This is contrasted with other solutions like Cypress, which provide their own testing framework and assertion library as part of an all-in-one solution.
- **An expressive but predictable API.** Puppeteer strikes a nice balance between low-level access to browser behavior, while retaining an expressive API for issuing and awaiting responses to those commands using modern JavaScript [async and await syntax](#). This is contrasted with other solutions, which either don't support or leverage native language async functionality, don't expose direct access to the browser, or leverage custom domain-specific language syntaxes for expressing browser commands and assertions. The fact that Puppeteer largely targets the Chrome browser is non-ideal in how it does not provide full browser coverage. On the other hand, the limited set of browser targets offers more consistent results and stronger guarantees about how code is evaluated in the browser environment.
- **Surfacing bugs, not obscuring them.** Many alternative solutions offer options to automatically await settled network requests or asynchronous appearance of elements on the page. While this can serve as a convenience in accounting for unpredictable delays, it can also unknowingly cause oversight of legitimate user-facing issues. For example, if an element will only appear on the page after some network request or computation has completed, it may be easy to overlook that these delays can cause unpredictable and frustrating behavior for users ([example](#)). Given that developers often test on high-end hardware and stable network connections, consideration of resiliency on low-end hardware

or spotty network availability is not always on the forefront of one's considerations. Puppeteer forces us to acknowledge these delays with explicit `waitFor*` expressions, putting us in much greater alignment with the real-world experience of an end-user.

- **Debugging.** It's important that in that case that a test fails, there should be straight-forward means to diagnose and resolve the issue. While its offerings are rather simplistic relative to the competition, Puppeteer does expose options to run tests as "headful" (with the browser visible) and with delayed actions. Combined with the fact that it interoperates well with native language / runtime features (e.g. debugger statements or breakpoints), this provides developers with sufficient debugging access.

For more context, refer to the following resources:

- [Testing Overview: End-to-End Testing](#)
- [Testing: Experiment with Puppeteer for E2E testing](#)
 - In early iterations, the contributing team opted to use Cypress for end-to-end testing. This pull request outlines problems with the approach, and proposed the initial transition to Puppeteer.
- [JavaScript Chat Summary: January 28, 2020](#)
 - Playwright is a new offering created by many of the original contributors to Puppeteer. It offers increased browser coverage and improved reliability of tests. While still early in development at the time of this writing, there has been some interest in evaluating it for future use as an end-to-end testing solution.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Automated Testing”](#)

[Previous Performance](#) [Previous: Performance](#)

[Next Site Editing Templates](#) [Next: Site Editing Templates](#)

Site Editing Templates

In this article

Table of Contents

- [Template and template part flows](#)
- [Storage](#)
- [Synchronization](#)
- [Switching themes](#)

[↑ Back to top](#)

Template and template part flows

This document will explain the internals of how templates and templates parts are rendered in the frontend and edited in the backend.

Storage

Just like the regular templates, the block templates live initially as files in the theme folder but the main difference is that the user can edit these templates in the UI in the Site Editor.

When a user edits a template (or template-part), the initial theme template file is kept as is but a forked version of the template is saved to the `wp_template` custom post type (or `wp_template_part` for template parts).

These capabilities mean that at any point in time, a mix of template files (from the theme) and CPT templates (the edited templates) are used to render the frontend of the site.

Synchronization

In order to simplify the algorithm used to edit and render the templates from two different places, we performed an operation called “template synchronization”.

The synchronization consists of duplicating the theme templates in the `wp_template` (and `wp_template_part`) custom templates with an `auto-draft` status. When a user edits these templates, the status is updated to `publish`.

This means:

- The rendering/fetching of templates only need to consider the custom post type templates. It is not necessary to fetch the template files from the theme folder directly. The synchronization will ensure these are duplicated in the CPT.
- Untouched theme templates have the `auto-draft` status.
- Edited theme templates have the `publish` status.

The synchronization is important for two different flows:

- When editing the template and template parts, the site editor frontend fetches the edited and available templates through the REST API. This means that for all GET API requests performed to the `wp-templates` and `wp-template-parts` endpoint synchronization is required.
- When rendering a template (sometimes referred to as “resolving a template”): this is the algorithm that WordPress follows to traverse the template hierarchy and find the right template to render for the current page being loaded.
- When exporting a block theme, we need to export all its templates back as files. The synchronization is required to simplify the operation and only export the CPT templates.

Switching themes

Since block themes make use of templates that can refer to each other and that can be saved to a custom post type, it becomes possible to mix templates and template parts from different themes. For example:

- A user might like the “header” template part of theme A and would like to use it in theme B.
- A user might like the “contact” template from theme A and would like to use it in theme B.

Enabling these flows will require well thought UIs and experience. For the current phase of Full-site editing, we’re starting by forbidding these possibilities and making template and template-parts theme specific.

That said, it is still important to keep track of where the template and template part come from initially. From which theme, it’s based. We do so by saving a `theme` post meta containing the theme identifier for each template and template part CPT entry.

In the future, we might consider allowing the user to mix template and template parts with different `theme` post meta values.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Site Editing Templates”](#)

[Previous Automated Testing](#) [Previous: Automated Testing](#)
[Next Styles in the Editor](#) [Next: Styles in the Editor](#)

Styles in the Editor

In this article

Table of Contents

- [HTML and CSS](#)
- [Block styles](#)
 - [From UI controls to HTML markup](#)
 - [Block Supports API](#)
 - [Current limitations of the Block Supports API](#)
- [Global styles](#)
 - [Gather data](#)
 - [Consolidate data](#)
 - [From data to styles](#)
 - [Current limitations of the Global Styles API](#)

- [Layout styles](#)
 - [Base layout styles](#)
 - [Individual layout styles](#)
 - [Available layout types](#)
 - [Targeting layout or container blocks from themes](#)
 - [Opting out of generated layout styles](#)

[↑ Back to top](#)

This document introduces the main concepts related to styles that affect the user content in the block editor. It points to the relevant reference guides and tutorials for readers to dig deeper into each one of the ideas presented. It's aimed to block authors and people working in the block editor project.

[HTML and CSS](#)

By creating a post in the block editor the user is creating a number of artifacts: a HTML document plus a number of CSS stylesheets, either embedded in the document or external.

The final HTML document is the result of a few things:

- the [WordPress templates](#) provided by the theme, either via PHP (classic theme) or via HTML templates (block theme) ([learn more](#) about the differences)
- the [blocks](#) and patterns in use that come with a predefined structure (HTML markup)
- the user modifications to the content: adding content, transforming existing content (convert a given paragraph into a heading), or modifying it (attaching a class or inline styles to a block)

The stylesheets loaded in the front end include:

- **Blocks.** The stylesheets that come with the block. In the front end, you can find a single stylesheet with all block styles defined by WordPress (`wp-block-library-*`) or separate stylesheets per block in use (as in `wp-block-group-*`, `wp-block-columns-*`, etc). See [this note](#) for the full details.
- **Global styles.** These styles are generated on the fly by using data coming from a `theme.json` file: see [note](#), [reference](#), and [how to guide](#). Specifically, it merges the contents of the `theme.json` from WordPress, the `theme.json` from the theme (if it has one), and the user data provided via the global styles sidebar in the site editor. The result of processing this data is an embedded stylesheet whose id is `global-styles-inline-css`.
- **Theme.** Historically, themes have enqueued their own stylesheets, where the id is based on the theme name, as in `twentytwentytwo-style.css`. In addition to having their own stylesheets they can now declare a `theme.json` file containing styles that will be part of the stylesheet generated by global styles.
- **User.** Some of the user actions in the editor will generate style content. This is the case for features such as duotone, layout, or link color.
- **Other.** WordPress and plugins can also enqueue stylesheets.

[Block styles](#)

Since the introduction of the block editor in WordPress 5.0, there were tools for the users to “add styles” to specific blocks. By using these tools, the user would attach new classes or inline styles to the blocks, modifying their visual aspect.

By default, blocks come with a given HTML markup. Think of the paragraph block, for example:

```
<p></p>
```

In its simplest form, any style rule that targets the p selector will apply styles to this block, whether it comes from a block, a theme, etc.

The user may change the state of this block by applying different styles: a text alignment, a color, a font size, a line height, etc. These states are reflected in the HTML markup of the block in the form of HTML attributes, mainly through the class or style attributes, though it can be any other the block author sees fit.

After some user modifications to the block, the initial markup may become something like this:

```
<p class="has-color has-green-color has-font-size has-small-font-size my-c  
style="line-height: 1em"></p>
```

This is what we refer to as “user-provided block styles”, also known as “local styles” or “serialized styles”. Essentially, each tool (font size, color, etc) ends up adding some classes and/or inline styles to the block markup. The CSS styling for these classes is part of the block, global, or theme stylesheets.

The ability to modify a block state coupled with the fact that a block can live within any other block (think of a paragraph within a group), creates a vast amount of potential states and style possibilities.

[From UI controls to HTML markup](#)

If you follow the [block tutorial](#) you can learn up about the different parts of the [block API](#) presented here in more detail and also build your own block. This is an introduction to the general concepts of how a block can let users edit its state.

To build an experience like the one described above a block author needs a few pieces:

1. **A UI control.** It presents the user some choices, for example, to be able to change the font size of the block. The control takes care of reading the data from the block (does this block already have a font size assigned?) and other data it needs (what are the font sizes a user can use in this block?). See available [component library](#).
2. **A block attribute.** The block needs to hold data to know which modifications were applied to it: whether it has been given a font size already for example. See how blocks can define [attributes](#).
3. **Access to style data.** A control may need external information about the styles available for a given block: the list of colors, or the list of font sizes, for example. These are called “style presets”, as they are a preselection of styles usually defined by the theme, although WordPress provides some defaults. Check the [list of data](#) a theme can provide to the editor and how a block author can get access to it via [useSetting](#).
4. **Serialize the user style into HTML markup.** Upon a user action, the block HTML markup needs to be updated accordingly (apply the proper class or inline style). This process is called serialization and it is the [edit](#), [save](#), and [render callback](#) functions’ responsibility: these functions take block data and convert it into HTML.

In essence, these are the essential mechanics a block author needs to care about for their block to be able to be styled by the user. While this can be done completely manually, there’s an API that automates this process for common style needs: block supports.

Block Supports API

[Block Supports](#) is an API that allows a block to declare what features it supports. By adding some info to their [block.json file](#), the block tells the system what kind of actions a user can do to it.

For example:

```
{  
    "name": "core/paragraph",  
    "...": "...",  
    "supports": {  
        "typography": {  
            "fontSize": true  
        }  
    }  
}
```

The paragraph declares support for font size in its `block.json`. This means the block will show a UI control for users to tweak its font size, unless it's disabled by the theme (learn more about how themes can disable UI controls in [the theme.json reference](#)). The system will also take care of setting up the UI control data (the font size of the block if it has one already assigned, the list of available font sizes to show), and will serialize the block data into HTML markup upon user changes (attach classes and inline styles appropriately).

By using the block supports mechanism via `block.json`, the block author is able to create the same experience as before just by writing a couple of lines. Check the tutorials for adding block supports to [static](#) and [dynamic](#) blocks.

Besides the benefit of having to do less work to achieve the same results, there's a few other advantages:

- the style information of the block becomes available for the native mobile apps and in the server
- the block will use the UI controls other blocks use for the same styles, creating a more coherent user experience
- the UI controls in use by the block will be automatically updated as they are improved, without the block author having to do anything

Current limitations of the Block Supports API

While the Block Supports API provides value, it also comes with some limitations a block author needs to be aware of. To better visualize what they are, let's run with the following example of a table block:

```
<table>  
  <thead>  
    <tr>  
      <th>Header</th>  
    </tr>  
  </thead>  
  <tbody>  
    <tr>  
      <th>First</th>  
    </tr>
```

```

<tr>
    <th>Second</th>
</tr>
</tbody>
<tfoot>
    <tr>
        <th>Footer</th>
    </tr>
</tfoot>
</table>

```

1. Only one style type per block.

One of the limitations is that, from all the [styles available](#), there can be only one instance of any them in use by the block. Following the example, the table block can only have a single font size. If the block author wanted to have three different font sizes (head, body, and footer) it can't do it using the current block supports API. See [this issue](#) for more detailed info and ways forward.

1. Styles are serialized to the outermost HTML node of the block, the wrapper.

The block supports API only serializes the font size value to the wrapper, resulting in the following HTML `<table class="has-small-font-size">`. The current block supports API doesn't serialize this value to a different node, for example, the `<tbody>`.

This is an active area of work you can follow [in the tracking issue](#). The linked proposal is exploring a different way to serialize the user changes: instead of each block support serializing its own data (for example, classes such as `has-small-font-size`, `has-green-color`) the idea is the block would get a single class instead (for example, `wp-style-UUID`) and the CSS styling for that class will be generated in the server by WordPress.

While work continues in that proposal, there's an escape hatch, an experimental option block authors can use. Any block support can skip the serialization to HTML markup by using `__experimentalSkipSerialization`. For example:

```
{
  "name": "core/paragraph",
  "...": "...",
  "supports": {
    "typography": {
      "fontSize": true,
      "__experimentalSkipSerialization": true
    }
  }
}
```

This means that the typography block support will do all of the things (create a UI control, bind the block attribute to the control, etc) except serializing the user values into the HTML markup. The classes and inline styles will not be automatically applied to the wrapper and it is the block author's responsibility to implement this in the `edit`, `save`, and `render_callback` functions. See [this issue](#) for examples of how it was done for some blocks provided by WordPress.

Note that, if `__experimentalSkipSerialization` is enabled for a group (typography, color, spacing) it affects *all* block supports within this group. In the example above *all* the

properties within the `typography` group will be affected (e.g. `fontSize`, `lineHeight`, `fontFamily` .etc).

To enable for a *single* property only, you may use an array to declare which properties are to be skipped. In the example below, only `fontSize` will skip serialization, leaving other items within the `typography` group (e.g. `lineHeight`, `fontFamily` .etc) unaffected.

```
{  
    "name": "core/paragraph",  
    "...": "...",  
    "supports": {  
        "typography": {  
            "fontSize": true,  
            "lineHeight": true,  
            "__experimentalSkipSerialization": [ "fontSize" ]  
        }  
    }  
}
```

Support for this feature was [added in this PR](#).

Global styles

Global Styles refers to a mechanism that generates site-wide styles. Unlike the block styles described in the previous section, these are not serialized into the post content and are not attached to the block HTML. Instead, the output of this system is a new stylesheet with id `global-styles-inline-css`.

This mechanism was [introduced in WordPress 5.8](#). At the time, it only took data from WordPress and the active theme. WordPress 5.9 expanded the system to also take style data from users.

This is the general data flow:

INPUT

WordPress'
theme.json

INTERNALS

Internal Represe

The process of generating the stylesheet has, in essence, three steps:

1. Gather data: the `theme.json` file [bundled with WordPress](#), the `theme.json` file of the active theme if it exists, and the user's styles provided via the global styles UI in the site editor.
2. Consolidate data: the structured information from different origins -WordPress defaults, theme, and user- is normalized and merged into a single structure.
3. Convert data into a stylesheet: convert the internal representation into CSS style rules and enqueue them as a stylesheet.

Gather data

The data can come from three different origins: WordPress defaults, the active theme, or the user. All three of them use the same [theme.json format](#).

Data from WordPress and the active theme is retrieved from the corresponding `theme.json` file. Data from the user is pulled from the database, where it's stored after the user saves the changes they did via the global styles sidebar in the site editor.

Consolidate data

The goal of this phase is to build a consolidated structure.

There are two important processes going on in this phase. First, the system needs to normalize all the incoming data, as different origins may be using different versions of the `theme.json` format. For example, a theme may be using [v1](#) while the WordPress base is using [the latest version](#). Second, the system needs to decide how to merge the input into a single structure. This will be the focus of the following sections.

Styles

Different parts of the incoming `theme.json` structure are treated differently. The data present in the `styles` section is blended together following this logic: user data overrides theme data, and theme data overrides WordPress data.

For example, if we had the following three `theme.json` structures coming from WordPress, the theme, and the user respectively:

```
{  
  "styles": {  
    "color": {  
      "background": "<WordPress value>"  
    },  
    "typography": {  
      "fontSize": "<WordPress value>"  
    }  
  }  
  
{  
  "styles": {  
    "typography": {  
      "fontSize": "<theme value>",  
      "lineHeight": "<theme value>"  
    }  
  }  
}
```

```

        }
    }

{
    "styles": {
        "typography": {
            "lineHeight": "<user value>"
        }
    }
}

```

The result after the consolidation would be:

```

{
    "styles": {
        "color": {
            "background": "<WordPress value>"
        },
        "typography": {
            "fontSize": "<theme value>",
            "lineHeight": "<user value>"
        }
    }
}

```

Settings

The `settings` section works differently than `styles`. Most of the settings are only used to configure the editor and have no effect on the global styles. Only a few of them are part of the resulting stylesheet: the `presets`.

Presets are the predefined styles that are shown to the user in different parts of the UI: the color palette or the font sizes, for example. They comprise the following settings: `color.duotone`, `color.gradients`, `color.palette`, `typography.fontFamilies`, `typography.fontSize`. Unlike `styles`, presets from an origin don't override values from other origins. Instead, all of them are stored in the consolidated structure.

For example, if we have the following `theme.json` structures coming from WordPress, the theme, and the user respectively:

```

{
    "settings": {
        "color": {
            "palette": [ "<WordPress values>" ],
            "gradients": [ "<WordPress values>" ]
        }
    }
}

{
    "settings": {
        "color": {
            "palette": [ "<theme values>" ]
        }
    }
}

```

```

        },
        "typography": {
            "fontFamilies": [ "<theme values>" ]
        }
    }
}

{
    "settings": {
        "color": {
            "palette": [ "<user values>" ]
        }
    }
}

```

The result after the consolidation would be:

```

{
    "settings": {
        "color": {
            "palette": {
                "default": [ "<WordPress values>" ],
                "theme": [ "<theme values>" ],
                "user": [ "<user values>" ]
            },
            "gradients": {
                "default": [ "<WordPress values>" ]
            }
        },
        "typography": {
            "fontFamilies": {
                "theme": [ "<theme values>" ]
            }
        }
    }
}

```

From data to styles

The last phase of generating the stylesheet is converting the consolidated data into CSS style rules.

Styles to CSS rules

The `styles` section can be thought of as a structured representation of CSS rules, each chunk representing a CSS rule:

- A key/value in `theme.json` maps to a CSS declaration (`property: value`).
- The CSS selector for a given chunk is generated based on its semantics:
 - The top-level section uses the `body` selector.
 - The top-level elements use an ID selector matching the HTML element they represent (for example, `h1` or `a`).

- Blocks use the default class name they generate (`core/group` becomes `.wp-block-group`) unless they explicitly set a different one using their `block.json` (`core/paragraph` becomes `p`). See the “Current limits” section for more about this.
- Elements within a block use the concatenation of the block and element selector.

For example, the following `theme.json` structure:

```
{
  "styles": {
    "typography": {
      "fontSize": "<top-level value>"
    },
    "elements": {
      "h1": {
        "typography": {
          "fontSize": "<h1 value>"
        }
      }
    },
    "blocks": {
      "core/paragraph": {
        "color": {
          "text": "<paragraph value>"
        }
      },
      "core/group": {
        "color": {
          "text": "<group value>"
        },
        "elements": {
          "h1": {
            "color": {
              "text": "<h1 within group value>"
            }
          }
        }
      }
    }
  }
}
```

is converted to the following CSS:

```
body {
  font-size: <top-level value>;
}
h1 {
  font-size: <h1 value>;
}
p {
  color: <paragraph value>;
}
.wp-block-group {
```

```

        color: <group value>;
    }
.wp-block-group h1 {
    color: <h1 within group value>;
}

```

Settings to CSS rules

From the `settings` section, all the values of any given presets will be converted to a CSS Custom Property that follows this naming structure: `--wp--preset--<category>-<slug>`. The selectors follow the same rules described in the styles section above.

For example, the following `theme.json`

```
{
    "settings": {
        "color": {
            "palette": {
                "default": [
                    {
                        "slug": "vivid-red",
                        "value": "#cf2e2e",
                        "name": "Vivid Red"
                    }
                ],
                "theme": [
                    {
                        "slug": "foreground",
                        "value": "#000",
                        "name": "Foreground"
                    }
                ]
            }
        },
        "blocks": {
            "core/site-title": {
                "color": {
                    "palette": {
                        "theme": [
                            {
                                "slug": "foreground",
                                "value": "#1a4548",
                                "name": "Foreground"
                            }
                        ]
                    }
                }
            }
        }
    }
}
```

Will be converted to the following CSS style rule:

```

body {
    --wp--preset--color--vivid-red: #cf2e2e;
    --wp--preset--color--foreground: #000;
}

.wp-block-site-title {
    --wp--preset--color--foreground: #1a4548;
}

```

In addition to the CSS Custom Properties, all presets but duotone generate CSS classes for each value. The example above will generate the following CSS classes as well:

```

/* vivid-red */
.has-vivid-red-color { color: var(--wp--preset--color--vivid-red) !important; }
.has-vivid-red-background-color { background-color: var(--wp--preset--color--vivid-red) !important; }
.has-vivid-red-border-color { border-color: var(--wp--preset--color--vivid-red) !important; }

/* foreground */
.has-foreground-color { color: var(--wp--preset--color--foreground) !important; }
.has-foreground-background-color { background-color: var(--wp--preset--color--foreground) !important; }
.has-foreground-border-color { border-color: var(--wp--preset--color--foreground) !important; }

/* foreground within site title */
.wp-block-site-title .has-foreground-color { color: var(--wp--preset--color--foreground) !important; }
.wp-block-site-title .has-foreground-background-color { background-color: var(--wp--preset--color--foreground) !important; }
.wp-block-site-title .has-foreground-border-color { border-color: var(--wp--preset--color--foreground) !important; }

```

[Current limitations of the Global Styles API](#)

1. Setting a different CSS selector for blocks requires server-registration

By default, the selector assigned to a block is `.wp-block-<block-name>`. However, blocks can change this should they need. They can provide a CSS selector via the `__experimentalSelector` property in its `block.json`.

If blocks do this, they need to be registered in the server using the `block.json`, otherwise, the global styles code doesn't have access to that information and will use the default CSS selector for the block.

2. Can't target different HTML nodes for different styles

Every chunk of styles can only use a single selector.

This is particularly relevant if the block is using `__experimentalSkipSerialization` to serialize the different style properties to different nodes other than the wrapper. See “Current limitations of blocks supports” for more.

3. Only a single property per block

Similarly to block supports, there can be only one instance of any style in use by the block. For example, the block can only have a single font size. See related “Current limitations of block supports”.

4. Only blocks using block supports are shown in the Global Styles UI

The global styles UI in the site editor has a screen for per-block styles. The list of blocks is generated dynamically using the block supports from the `block.json` of blocks. If a block wants to be listed there, it needs to use the block supports mechanism.

Layout styles

In addition to styles at the individual block level and in global styles, there is the concept of layout styles that are output for both blocks-based and classic themes.

The layout block support outputs common layout styles that are shared between blocks used for creating layouts. Layout styles are useful for providing common styling for any block that is a container for other blocks. Examples of blocks that depend on these layout styles include Group, Row, Columns, Buttons, and Social Icons. The feature is enabled in core blocks via the `layout` setting under `supports` in a block's `block.json` file.

There are two primary places where Layout styles are output:

Base layout styles

Base layout styles are those styles that are common to all blocks that opt in to a particular layout type. Examples of common base layout styling include setting `display: flex` for blocks that use the Flex layout type (such as Buttons and Social Icons), and providing default max-width for constrained layouts.

Base layout styles are output from within [the main PHP class](#) that handles global styles, and form part of the global styles stylesheet. In order to provide support for core blocks in classic themes, these styles are always output, irrespective of whether the theme provides its own `theme.json` file.

Common layout definitions are stored in [the core layout block support file](#).

Individual layout styles

When a block that opts in to layout support is rendered, two things are processed and added to the output via [layout.php](#):

- Semantic class names are added to the block markup to indicate which layout settings are in use. For example, `is-layout-flow` is for blocks (such as Group) that use the default/flow layout, and `is-content-justification-right` is added when a user sets a block to use right justification.
- Individual styles are generated for non-default layout values that are set on the individual block being rendered. These styles are attached to the block via a container class name using the form `wp-container-$id` where the `$id` is a [unique number](#).

Available layout types

There are currently four layout types in use:

- Default/Flow: Items are stacked vertically. The parent container block is set to `display: flow` and the spacing between children is handled via vertical margins.

- Constrained: Items are stacked vertically, using the same spacing logic as the Flow layout. Features constrained widths for child content, outputting widths for standard content size and wide size. Defaults to using global `contentSize` and `wideSize` values set in `settings.layout` in the `theme.json`.
- Flex: Items are displayed using a Flexbox layout. Defaults to a horizontal orientation. Spacing between children is handled via the `gap` CSS property.
- Grid: Items are displayed using a Grid layout. Defaults to an `auto-fill` approach to column generation but can also be set to a fixed number of columns. Spacing between children is handled via the `gap` CSS property.

For controlling spacing between blocks, and enabling block spacing controls see: [What is blockGap and how can I use it?](#)

Targeting layout or container blocks from themes

The layout block support is designed to enable control over layout features from within the block and site editors. Where possible, try to use the features of the blocks to determine particular layout requirements rather than relying upon additional stylesheets.

For themes that wish to target container blocks in order to add or adjust particular styles, the block's class name is often the best class name to use. Class names such as `wp-block-group` or `wp-block-columns` are usually reliable class names for targeting a particular block. In addition to block and layout classnames, there is also a classname composed of block and layout together: for example, for a Group block with a constrained layout it will be `wp-block-group-is-layout-constrained`.

For targeting a block that uses a particular layout type, avoid targeting `wp-container-` as container classes may not always be present in the rendered markup.

Semantic class names

Work is currently underway to expand stable semantic classnames in Layout block support output. The task is being discussed in [this issue](#).

The current semantic class names that can be output by the Layout block support are:

- `is-layout-flow`: Blocks that use the Default/Flow layout type.
- `is-layout-constrained`: Blocks that use the Constrained layout type.
- `is-layout-flex`: Blocks that use the Flex layout type.
- `is-layout-grid`: Blocks that used the Grid layout type.
- `wp-container-$id`: Where `$id` is a semi-random number. A container class that only exists when the block contains non-default Layout values. This class should not be used directly for any CSS targeting as it may or may not be present.
- `is-horizontal`: When a block explicitly sets `orientation` to `horizontal`.
- `is-vertical`: When a block explicitly sets `orientation` to `vertical`.
- `is-content-justification-left`: When a block explicitly sets `justifyContent` to `left`.
- `is-content-justification-center`: When a block explicitly sets `justifyContent` to `center`.
- `is-content-justification-right`: When a block explicitly sets `justifyContent` to `right`.
- `is-content-justification-space-between`: When a block explicitly sets `justifyContent` to `space-between`.
- `is-nowrap`: When a block explicitly sets `flexWrap` to `nowrap`.

[Opting out of generated layout styles](#)

Layout styles output is switched on by default because the styles are required by core structural blocks. However, themes can opt out of generated block layout styles while retaining semantic class name output by using the `disable-layout-styles` block support. Such themes will be responsible for providing all their own layout styles. See [the entry under Theme Support](#).

First published

February 3, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Styles in the Editor”](#)

[Previous Site Editing Templates](#) [Previous: Site Editing Templates](#)

[Next User Interface](#) [Next: User Interface](#)

User Interface

In this article

Table of Contents

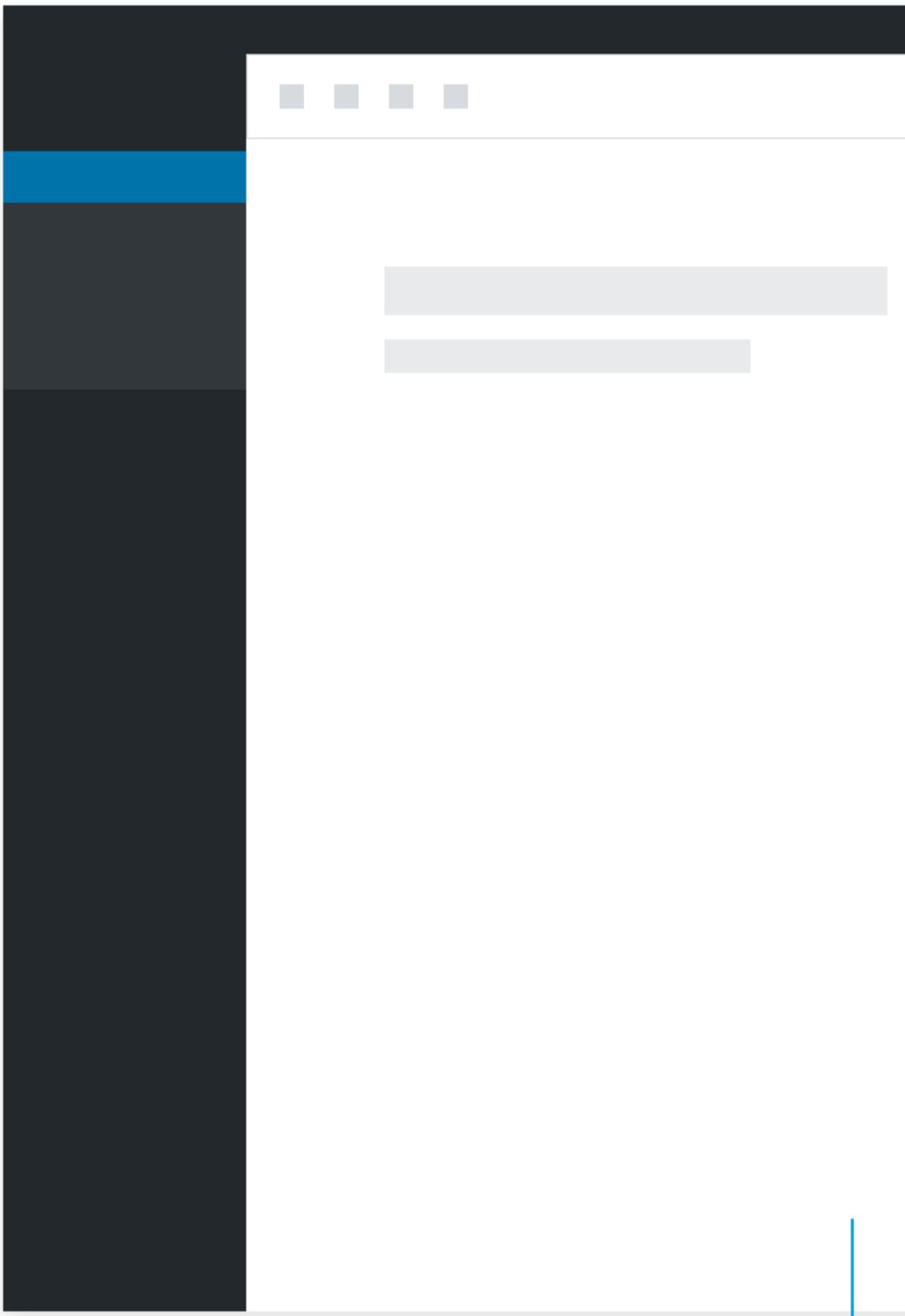
- [The Block Editor](#)
- [The Block](#)
- [Settings Sidebar](#)
- [Block Library](#)

[↑ Back to top](#)

[The Block Editor](#)

The block editor’s general layout uses a bar at the top, with content below.

Large Screens



Content Area

The **Toolbar** contains document-level actions: Editor>Select modes, save status, global actions for undo/redo/insert, the settings toggle, and publish options.

The **Content Area** contains the document itself.

The **Settings Sidebar** contains additional settings for the document (tags, categories, schedule etc.) and for blocks in the “Block” tab. A cog button in the toolbar hides the Settings Sidebar, allowing the user to enjoy a more immersive writing experience. On small screens, the sidebar is hidden by default.

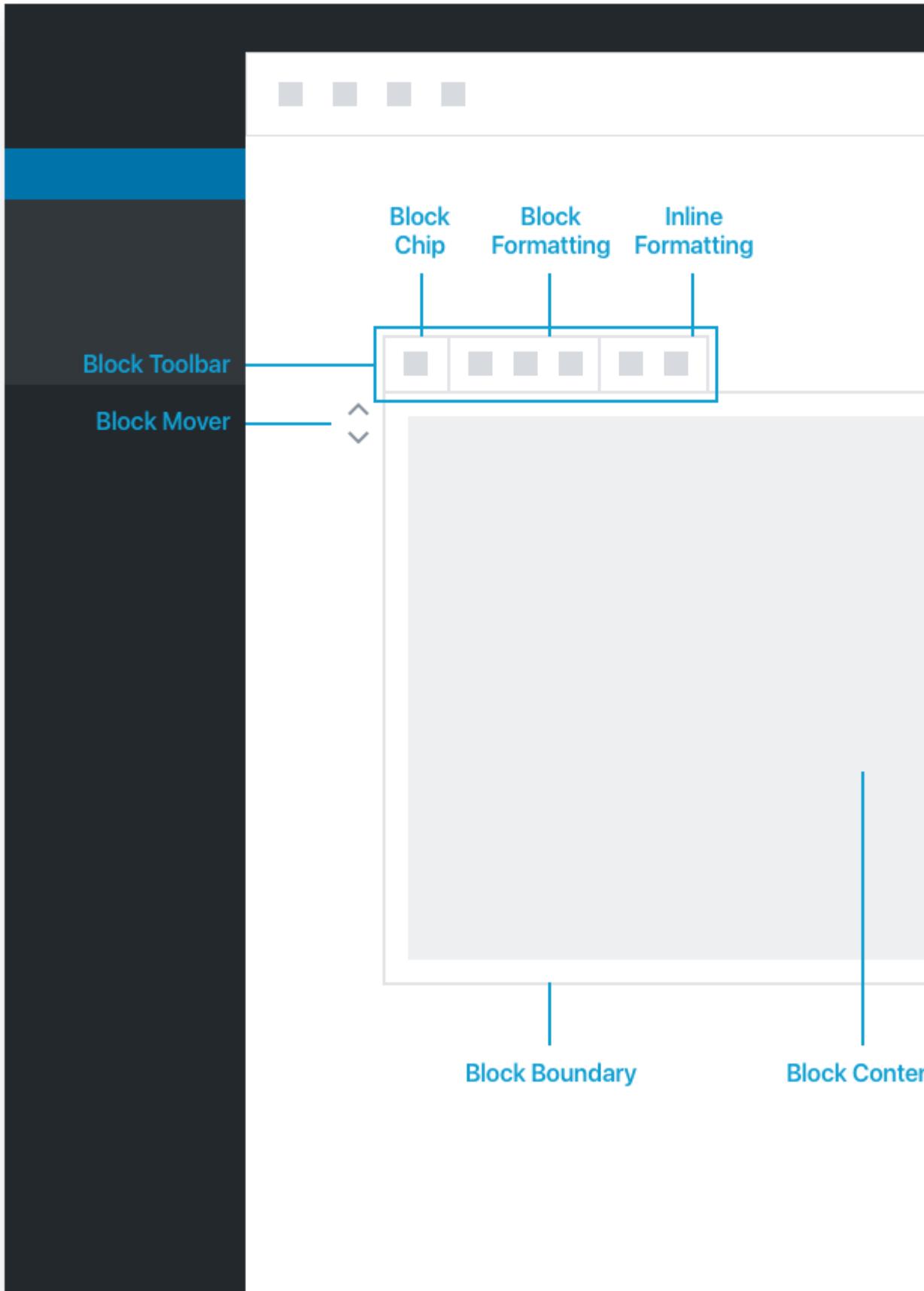
The Block

The block itself is the most basic unit of the editor. Generally speaking, everything is a block. Users build posts and pages using blocks, mimicking the vertical flow of the underlying HTML markup.

By surfacing each section of the document as a manipulatable block, we surface block-specific features contextually. This is inspired by desktop app conventions, and allows for a breadth of advanced features without weighing down the UI.

A selected block shows a number of contextual actions:

Large Screens



The block interface has basic actions. The block editor aims for good, common defaults, so users should be able to create a complete document without actually needing the advanced actions in the Settings Sidebar.

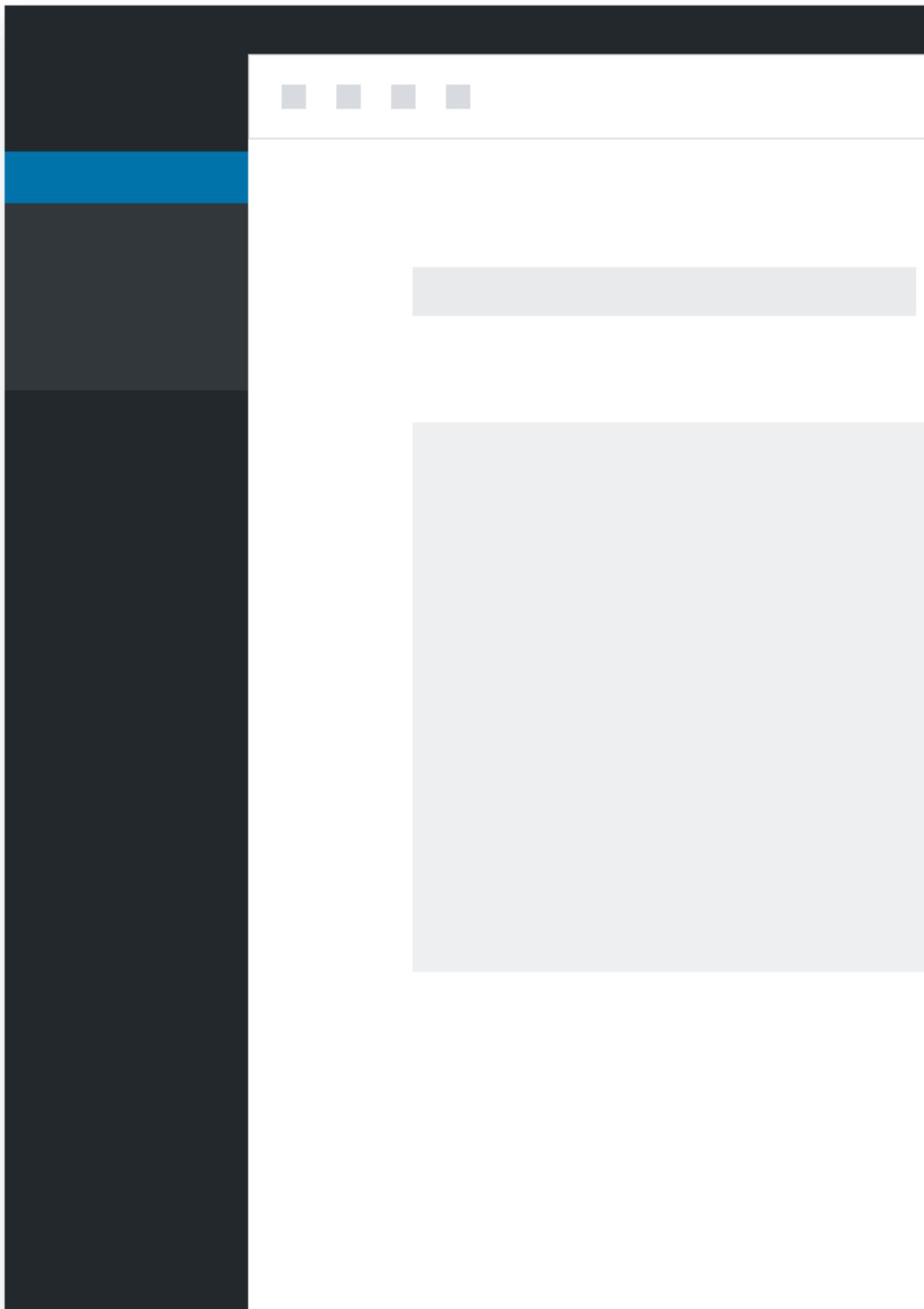
The **Block Toolbar** highlights commonly-used actions. The **Block Icon** lives in the block toolbar, and contains high-level controls for the selected block. It primarily allows users to transform a block into another type of compatible block. Some blocks also use the block icon for users to choose from a set of alternate block styles.

The **Block Formatting** options let users adjust block-level settings, and the **Inline Formatting** options allow adjustments to elements inside the block. When a block is long, the block toolbar pins itself to the top of the screen as the user scrolls down the page.

Blocks can be moved up and down via the **Block Mover** icons. Additional block actions are available via an ellipsis menu: deleting and duplicating blocks, as well as **advanced actions** like “Edit as HTML” and “Convert to Reusable Block.”

An unselected block does not show the block toolbar or any other contextual controls. In effect, an unselected block is a preview of the content itself:

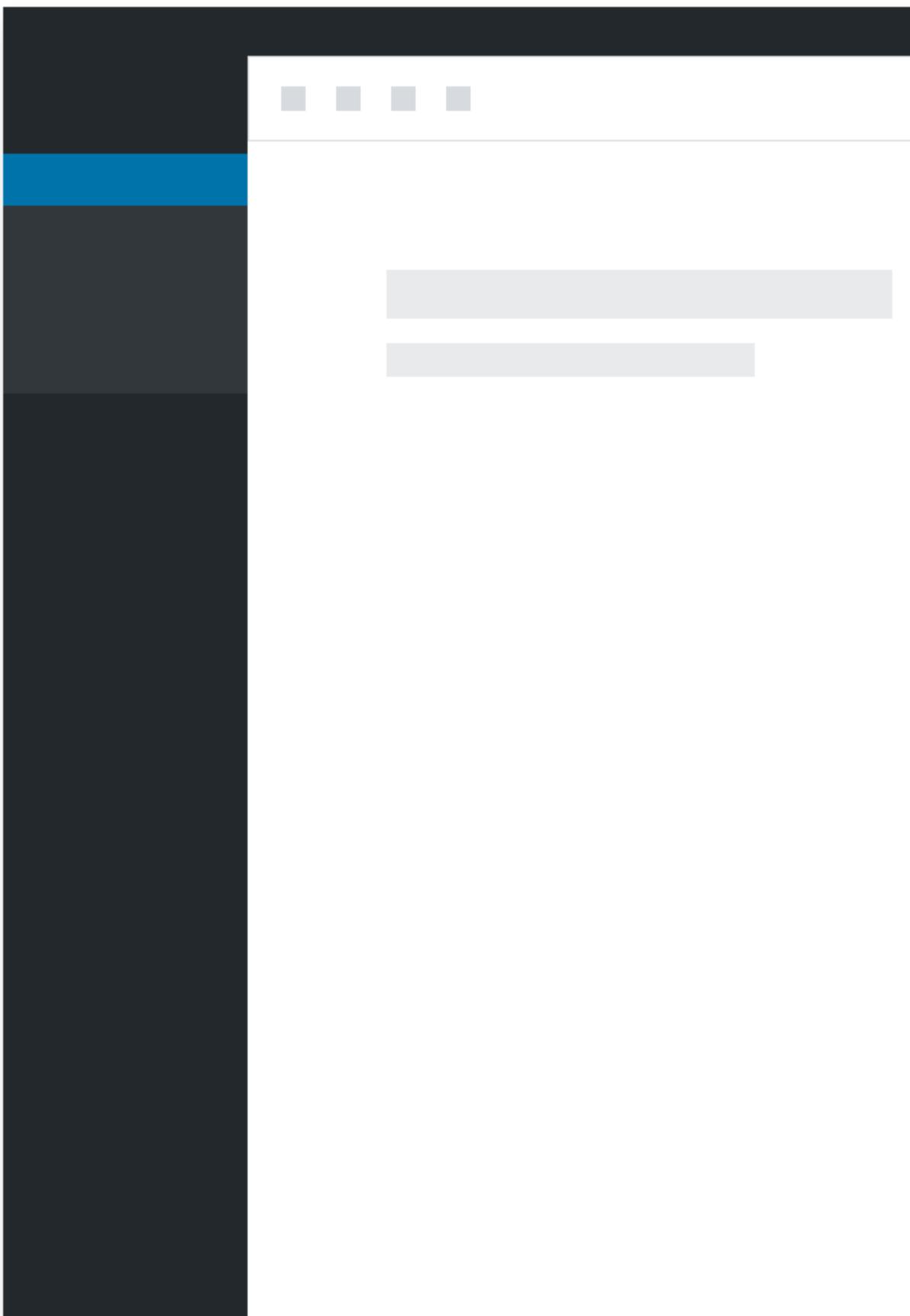
Large Screens



Please note that selection and focus can be different. An image block can be selected while the focus is on the caption field.

Settings Sidebar

Large Screens



The sidebar has two tabs, Document and Block:

- The **Document Tab** shows metadata and settings for the post or page being edited.
- The **Block Tab** shows metadata and settings for the currently selected block.

Each tab has sets of editable fields (**Sidebar Sections**) that users can toggle open or closed.

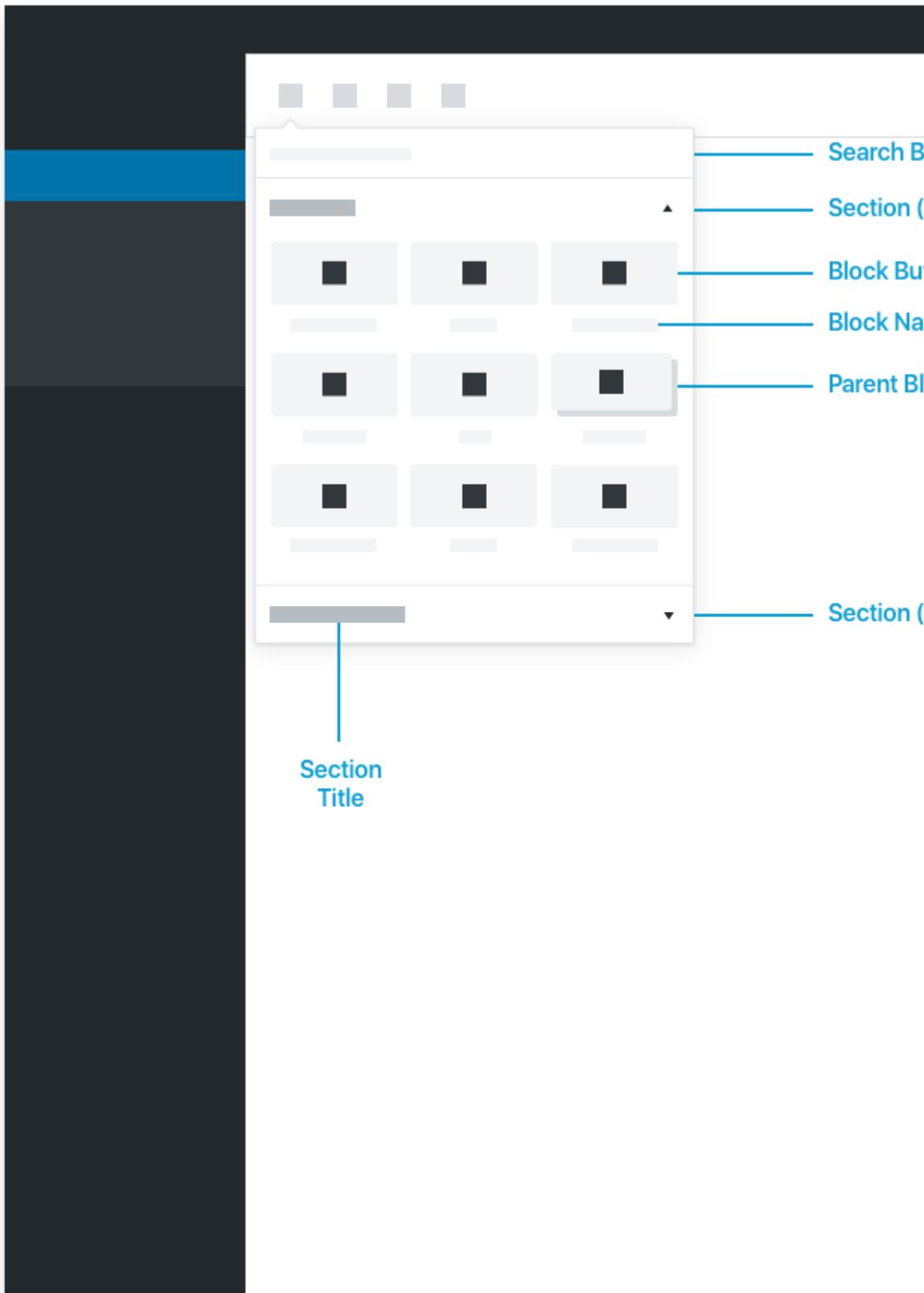
If a block requires advanced configuration, those settings should live in the Settings Sidebar. Don't put anything in the sidebar block tab that is necessary for the basic operation of your block; your user might dismiss the sidebar for an immersive writing experience. Pick good defaults, and make important actions available in the block toolbar.

Actions that could go in the block tab of the sidebar could be:

- Drop cap, for text
- Number of columns for galleries
- Number of posts, or category, in the “Latest Posts” block
- Any configuration that you don't need access to in order to perform basic tasks

Block Library

Large Screens



The **Block Library** appears when someone inserts a block, whether via the toolbar, or contextually within the content area. Inside, blocks are organized into expandable sections. The block library's search bar auto-filters the list of blocks as the user types. Users can choose a block by selecting the **Block Button** or the **Block Name**.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: User Interface”](#)

[Previous Styles in the Editor](#) [Previous: Styles in the Editor](#)

[Next Block Design](#) [Next: Block Design](#)

Block Design

In this article

Table of Contents

- [Best Practices](#)
 - [The primary interface for a block is the content area of the block](#)
 - [The Block Toolbar is a secondary place for required options & controls](#)
 - [Group Block Toolbar controls with related items](#)
 - [The Settings Sidebar should only be used for advanced, tertiary controls](#)
- [Setup state vs. live preview state](#)
- [Do's and Don'ts](#)
 - [Block Toolbar](#)
 - [Block identification](#)
 - [Block description](#)
 - [Placeholders](#)
 - [Selected and unselected states](#)
 - [Advanced block settings](#)
 - [Consider mobile](#)
 - [Support Gutenberg's dark background editor scheme](#)
- [Examples](#)
 - [Paragraph](#)
 - [Image](#)
 - [Latest Post](#)

[↑ Back to top](#)

The following are best practices for designing a new block, with recommendations and detailed descriptions of existing blocks to illustrate our approach to creating blocks.

Best Practices

The primary interface for a block is the content area of the block

Since the block itself represents what will actually appear on the site, interaction here hews closest to the principle of direct manipulation and will be most intuitive to the user. This should be thought of as the primary interface for adding and manipulating content and adjusting how it is displayed. There are two ways of interacting here:

1. The placeholder content in the content area of the block can be thought of as a guide or interface for users to follow a set of instructions or “fill in the blanks”. For example, a block that embeds content from a 3rd-party service might contain controls for signing in to that service in the placeholder.
2. After the user has added content, selecting the block can reveal additional controls to adjust or edit that content. For example, a signup block might reveal a control for hiding/showing subscriber count. However, this should be done in minimal ways, so as to avoid dramatically changing the size and display of a block when a user selects it (this could be disorienting or annoying).

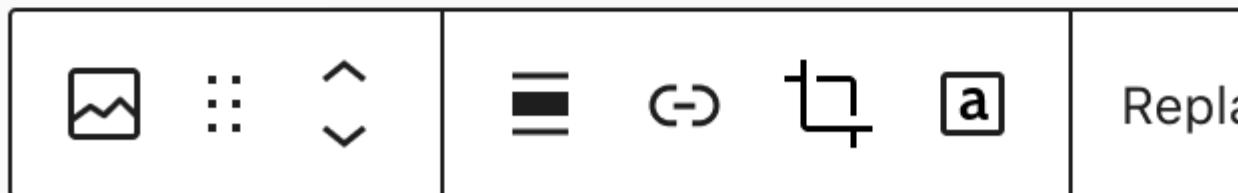
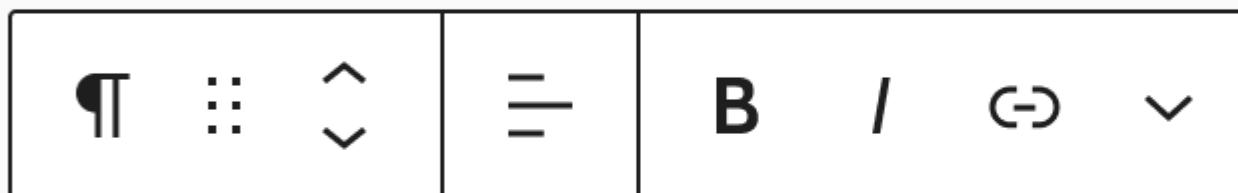
The Block Toolbar is a secondary place for required options & controls

Basic block settings won’t always make sense in the context of the placeholder/content UI. As a secondary option, options that are critical to the functionality of a block can live in the block toolbar. The Block Toolbar is still highly contextual and visible on all screen sizes. One notable constraint with the Block Toolbar is that it is icon-based UI, so any controls that live in the Block Toolbar need to be ones that can effectively be communicated via an icon or icon group.

Group Block Toolbar controls with related items

The Block Toolbar groups controls in segments, hierarchically. The first segment contains block type controls, such as the block switcher, the drag handle, and the mover control. The second group contains common and specific block tools that affect the entire block, followed by inline formatting, and the “More” menu. Optionally “Meta” or “Other” groups can separate some tools in their own segment.

Structure



The Settings Sidebar should only be used for advanced, tertiary controls

The Settings Sidebar is not visible by default on a small / mobile screen, and may also be collapsed in a desktop view. Therefore, it should not be relied on for anything that is necessary for the basic operation of the block. Pick good defaults, make important actions available in the

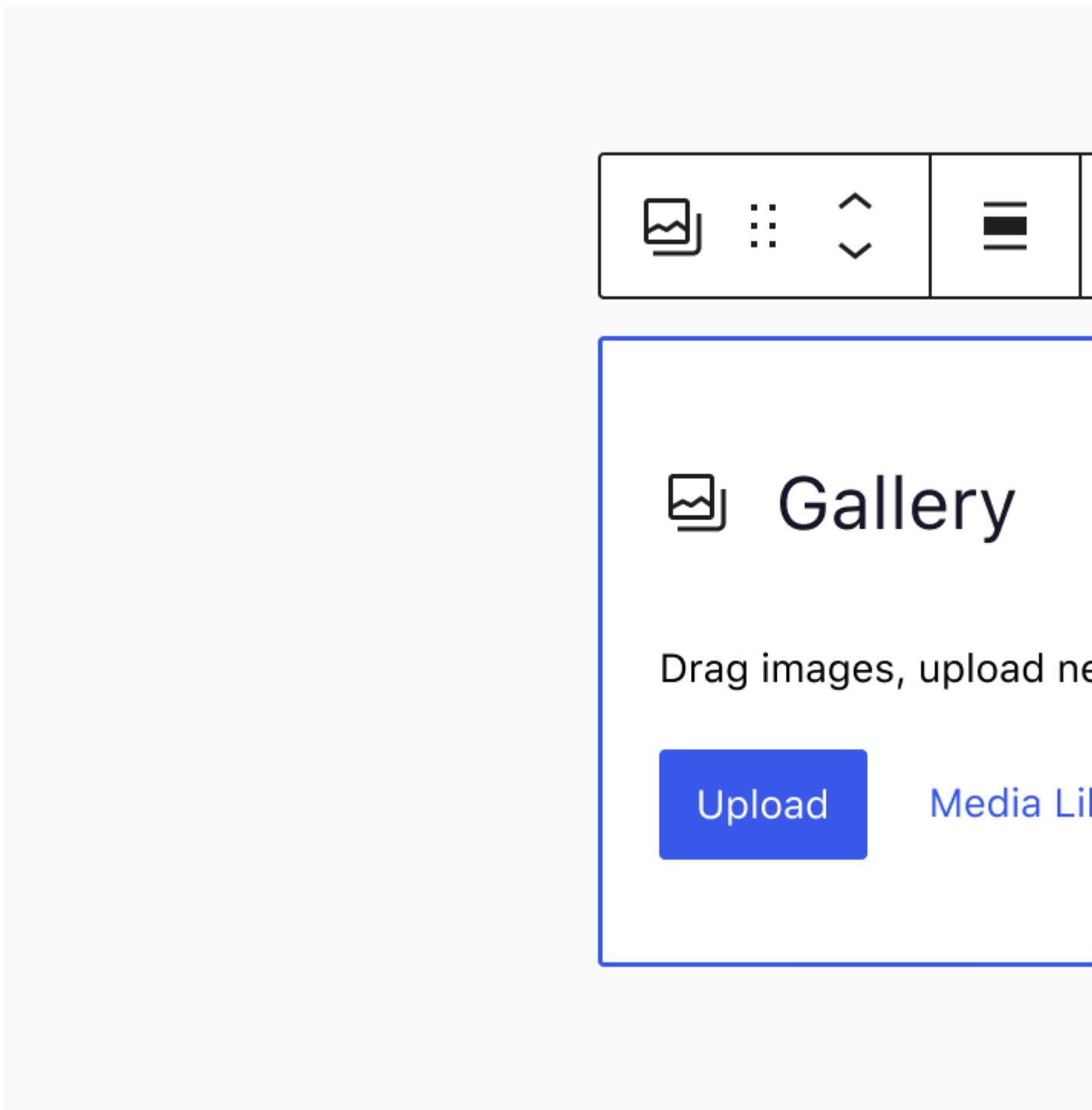
block toolbar, and think of the Settings Sidebar as something that most users should not need to open.

In addition, use sections and headers in the Settings Sidebar if there are more than a handful of options, in order to allow users to easily scan and understand the options available.

Each Settings Sidebar comes with an “Advanced” section by default. This area houses an “Additional CSS Class” field, and should be used to house other power user controls.

Setup state vs. live preview state

Setup states, sometimes referred to as “placeholders”, can be used to walk users through an initial process before showing the live preview state of the block. The setup process gathers information from the user that is needed to render the block. A block’s setup state is indicated with a grey background to provide clear differentiation for the user. Not all blocks have setup states — for example, the Paragraph block.



A setup state is **not** necessary if:

- You can provide good default content in the block that will meet most people's needs.
- That default content is easy to edit and customize.

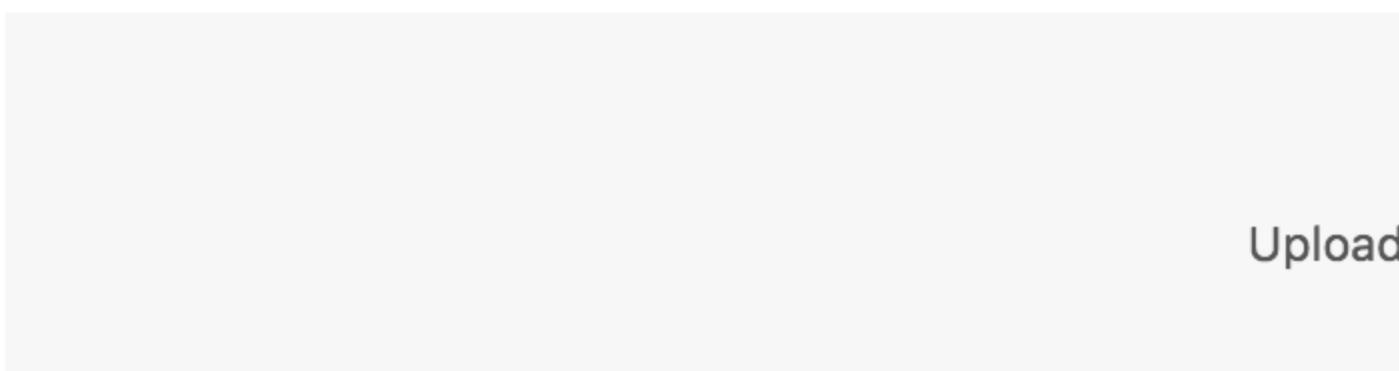
Use a setup state if:

- There isn't a clear default state that would work for most users.
- You need to gather input from the user that doesn't have a 1-1 relationship with the live preview of the block (for example, if you need the user to input an API key to render content).
- You need more information from the user in order to render useful default content.

For blocks that do have setup states, once the user has gone through the setup process, the placeholder is replaced with the live preview state of that block.



When the block is selected, additional controls may be revealed to customize the block's contents. For example, when the image gallery is selected, it reveals controls to remove or add images.



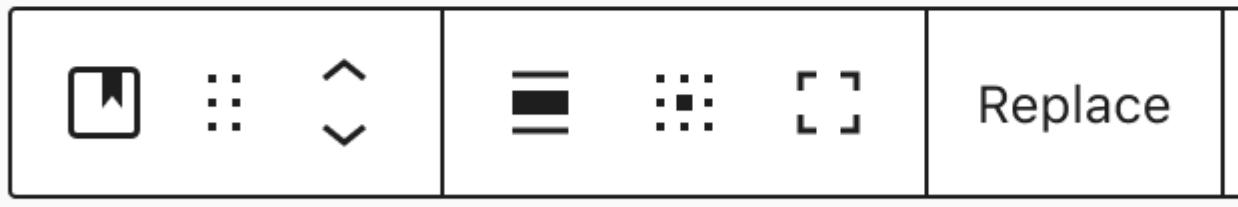
In most cases, a block's setup state is only shown once and then further customization is done via the live preview state. However, in some cases it might be desirable to allow the user to return to the setup state — for example, if all the block content has been deleted or via a link from the block's toolbar or sidebar.

Do's and Don'ts

Block Toolbar

Group toolbar controls in logical segments. Don't add a segment for each.

Do ✓



Block identification

A block should have a straightforward, short name so users can easily find it in the block library. A block named “YouTube” is easy to find and understand. The same block, named “Embedded Video (YouTube)”, would be less clear and harder to find in the block library.

When referring to a block in documentation or UI, use title case for the block title and lowercase for the “block” descriptor. For example:

- Paragraph block
- Latest Posts block
- Media & Text block

Blocks should have an identifying icon, ideally using a single color. Try to avoid using the same icon used by an existing block. The core block icons are based on [Material Design Icons](#). Look to that icon set, or to [Dashicons](#) for style inspiration.



Image



Gallery

Do:

Use concise block names.



Acme Photo
Slideshow



Wonka
Industries
Contact Form

Don't:

Avoid long, multi-line block names.

Block description

Every block should include a description that clearly explains the block's function. The description will display in the Settings Sidebar.

You can add a description by using the [description attribute in the registerBlockType function](#).

Stick to a single imperative sentence with an action + subject format. Examples:

- Start with the basic building block of all narrative.
- Introduce new sections and organize content to help visitors (and search engines) understand the structure of your content.
- Create a bulleted or numbered list.



Google Map

Show a map from a Google Maps share URL.

Do:

Use a short, simple block description.



Google Map

If you'd like to embed a Google Map, this block can help you! Paste it in the "Google Map URL:" field.

An Acme™ Block.
Feel the breeze!

Don't:

Avoid long descriptions and branding.

Placeholders

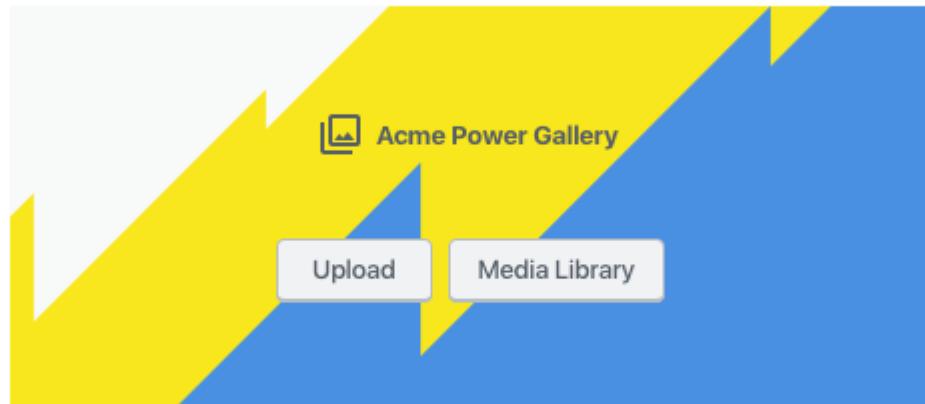
If your block requires a user to configure some options before you can display it, you should provide an instructive placeholder state.



The placeholder state shows a large, light gray rectangular area with rounded corners. In the center, there is a small icon of a photo frame and the word "Gallery". Below this, the text "Drag images, upload a new one, or select a file from your library." is displayed. At the bottom, there are two buttons: "Upload" on the left and "Media Library" on the right, both enclosed in thin rectangular borders.

Do:

Provide an instructive placeholder state.



Don't:

Avoid branding and relying on the title alone to convey instructions.

Selected and unselected states

When unselected, your block should preview its content as closely to the front-end output as possible.

When selected, your block may surface additional options like input fields or buttons to configure the block directly, especially when they are necessary for basic operation.



Unselected state:

Visit Easter Island!



Selected state:



Google Map URL:

<https://goo.gl/maps/VnNK3iXqv9A2>

Do:

For controls that are essential for the operation of the block, provide them directly inside the block edit view.

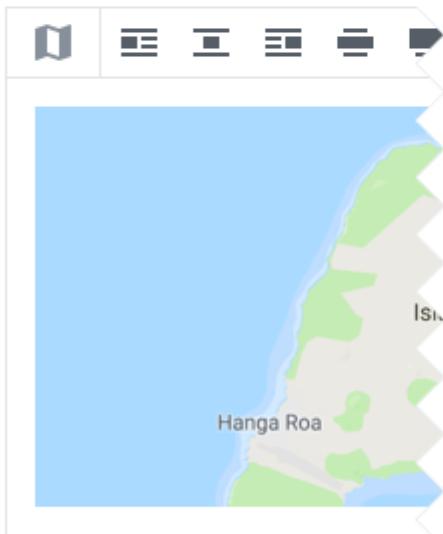


Unselected state:

Visit Easter Island!



Selected state:



Document

Block



Google Map

Show a map from a Google Maps share URL.

Google Map URL

`https://goo.gl/maps/VnNK3iXqv9A2`

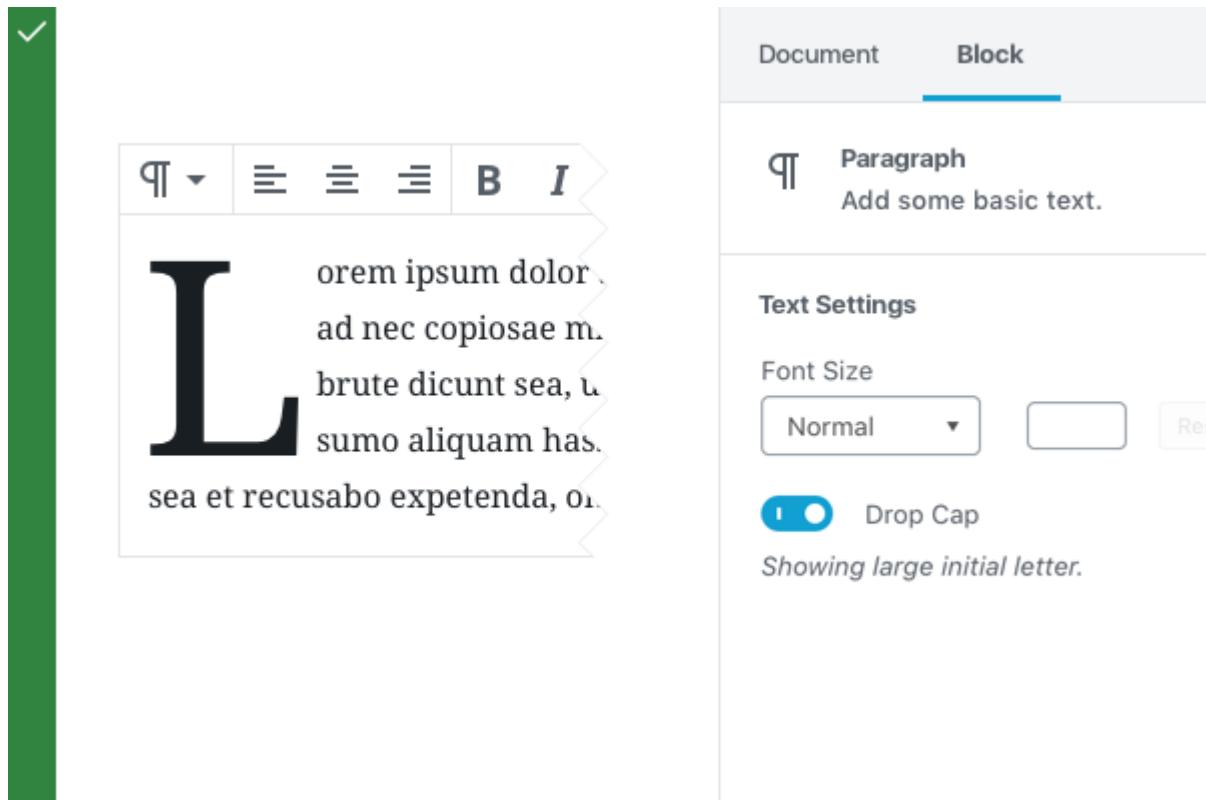
Additional Settings

Don't:

Do not put controls that are essential to the block in the sidebar, otherwise the block will appear non-functional to mobile users or desktop users who have dismissed the sidebar.

[Advanced block settings](#)

The “Block” tab of the Settings Sidebar can contain additional block options and configuration. Keep in mind that a user can dismiss the sidebar and never use it. You should not put critical options in the Sidebar.



Do:

Because the Drop Cap feature is not necessary for the basic operation of the block, you can put it to the Block tab as optional configuration.

Consider mobile

Check how your block looks, feels, and works on as many devices and screen sizes as you can.

Support Gutenberg's dark background editor scheme

Check how your block looks with [dark backgrounds](#) in the editor.

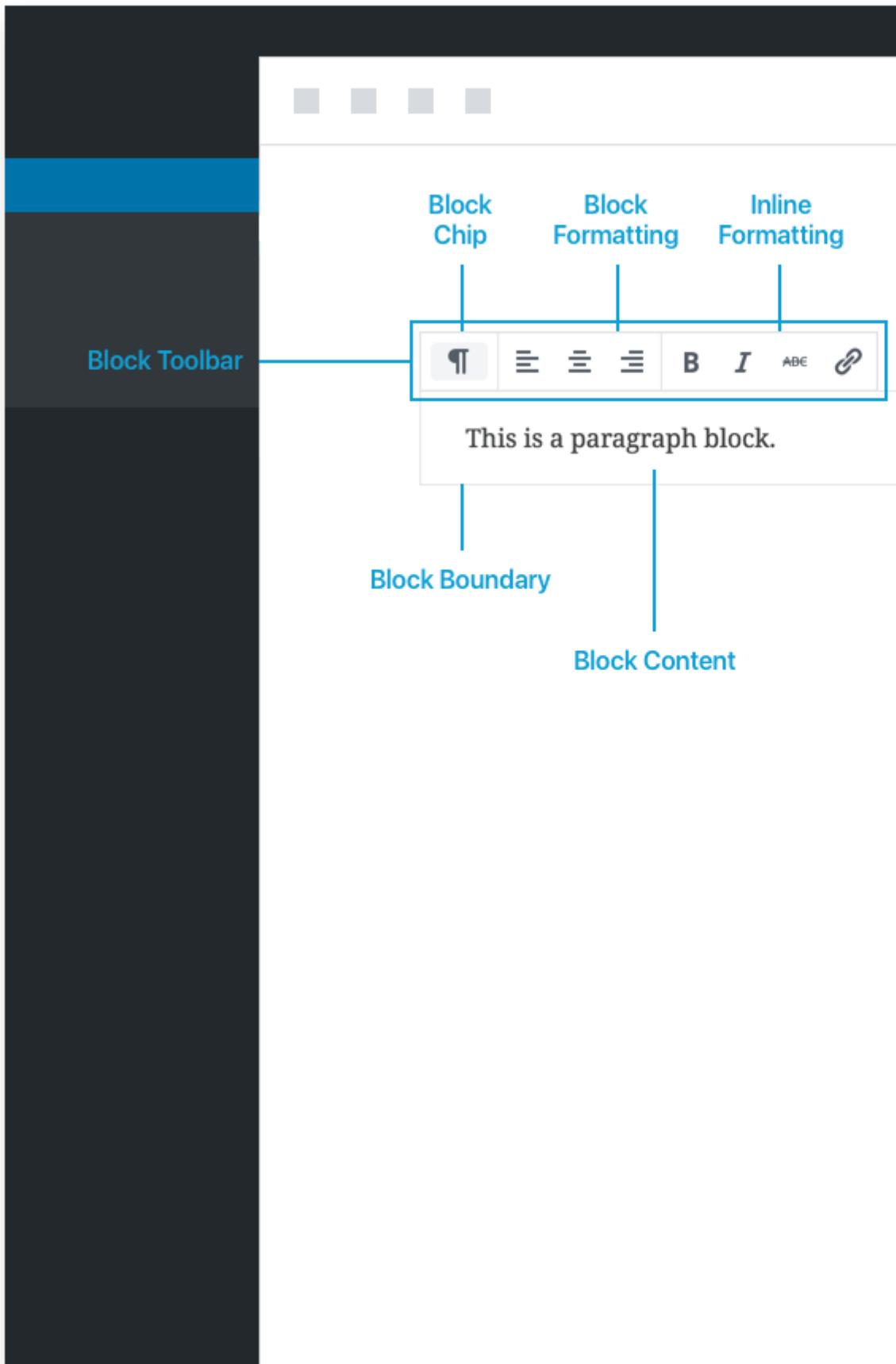
Examples

To demonstrate some of these practices, here are a few annotated examples of default Gutenberg blocks:

Paragraph

The most basic unit of the editor. The Paragraph block is a simple input field.

Large Screens



Placeholder

- Simple placeholder text that reads “Type / to choose a block”. The placeholder disappears when the block is selected.

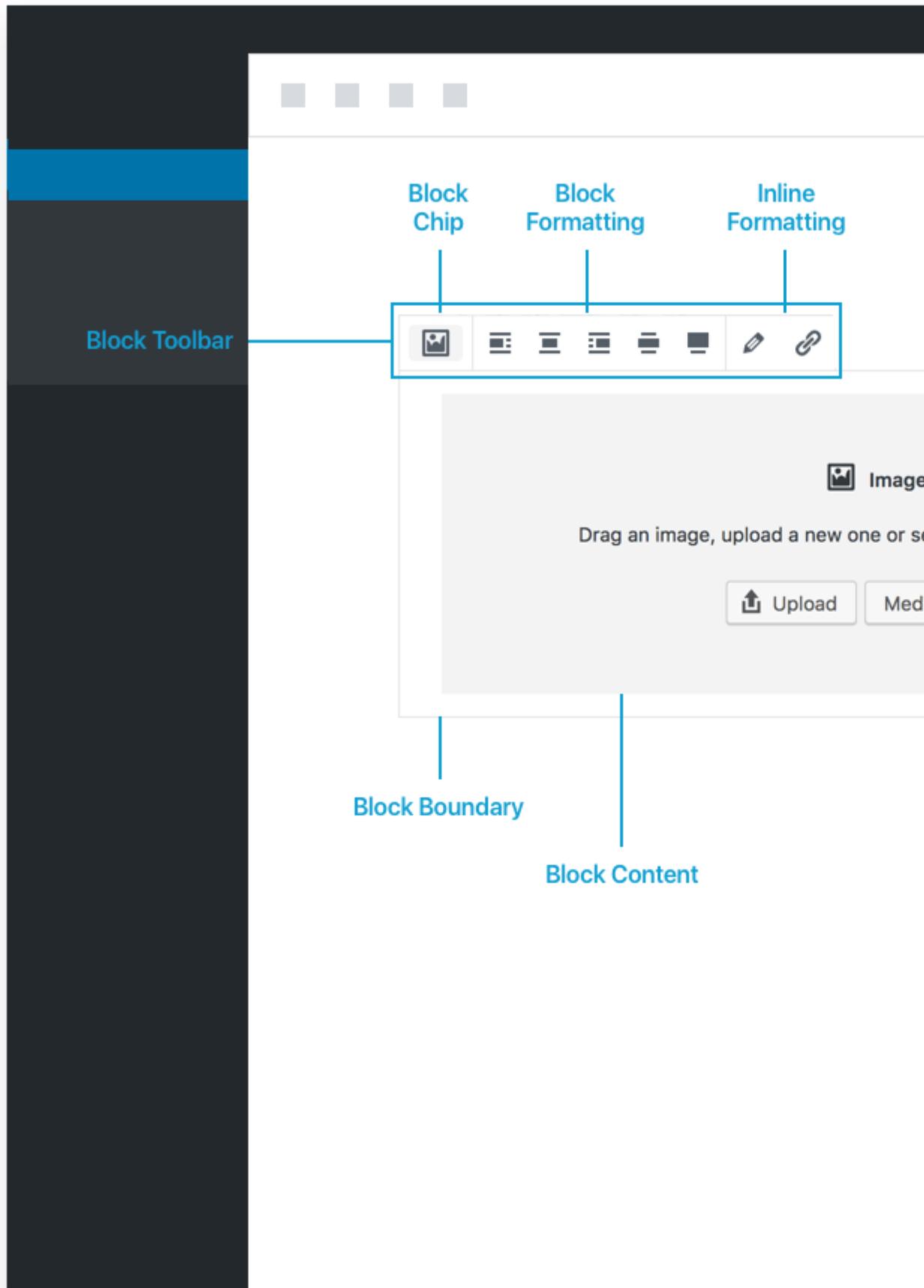
Selected state

- Block Toolbar: Has a switcher to perform transformations to headings, etc.
- Block Toolbar: Has basic text alignments
- Block Toolbar: Has inline formatting options, bold, italic, strikethrough, and link

Image

Basic image block.

Large Screens



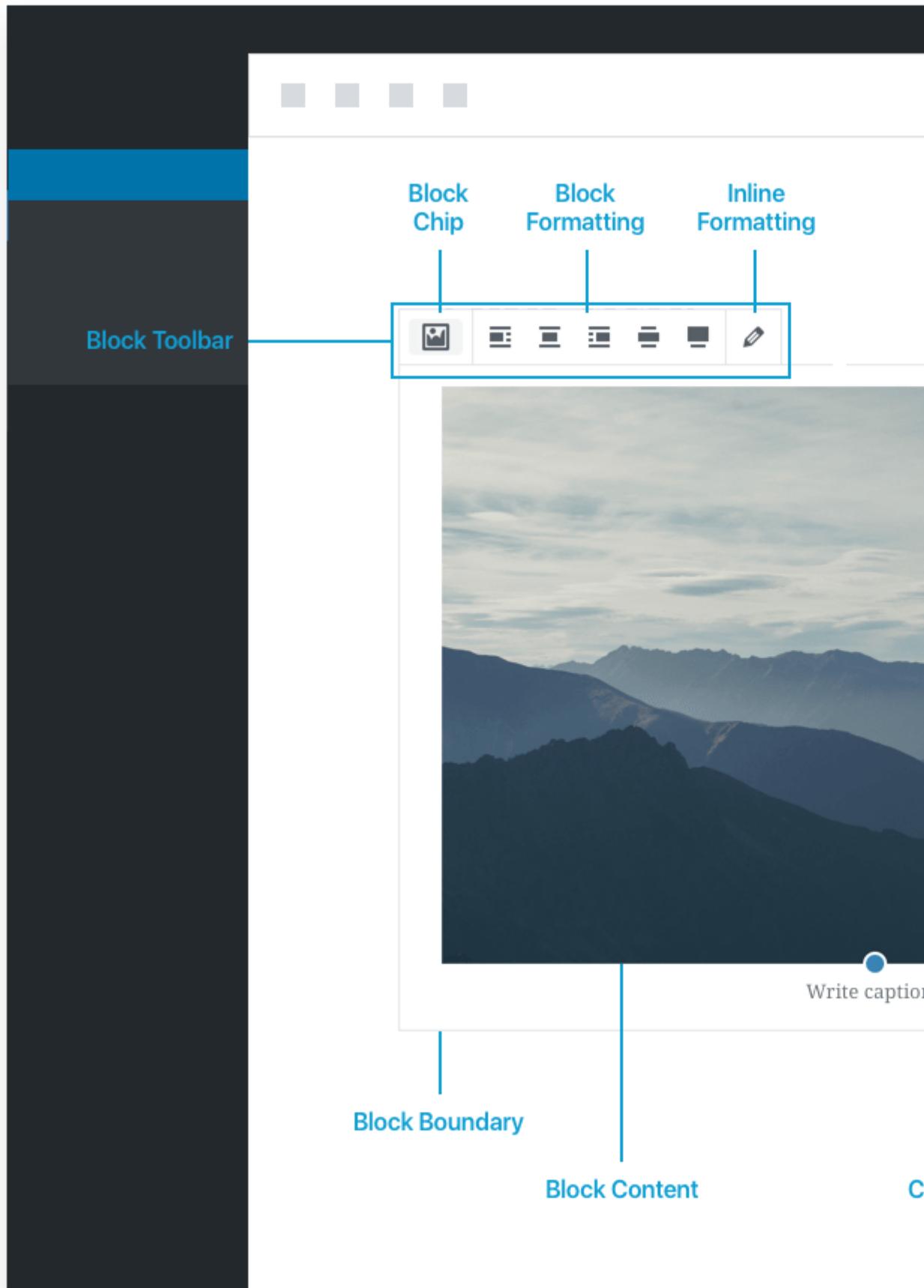
Placeholder

- A generic gray placeholder block with options to upload an image, drag and drop an image directly on it, or pick an image from the media library.

Selected state

- Block Toolbar: Alignments, including wide and full-width if the theme supports it.
- Block Toolbar: Edit Image, to open the Media Library
- Block Toolbar: Link button
- When an image is uploaded, a caption input field appears with a “Write caption...” placeholder text below the image:

Large Screens



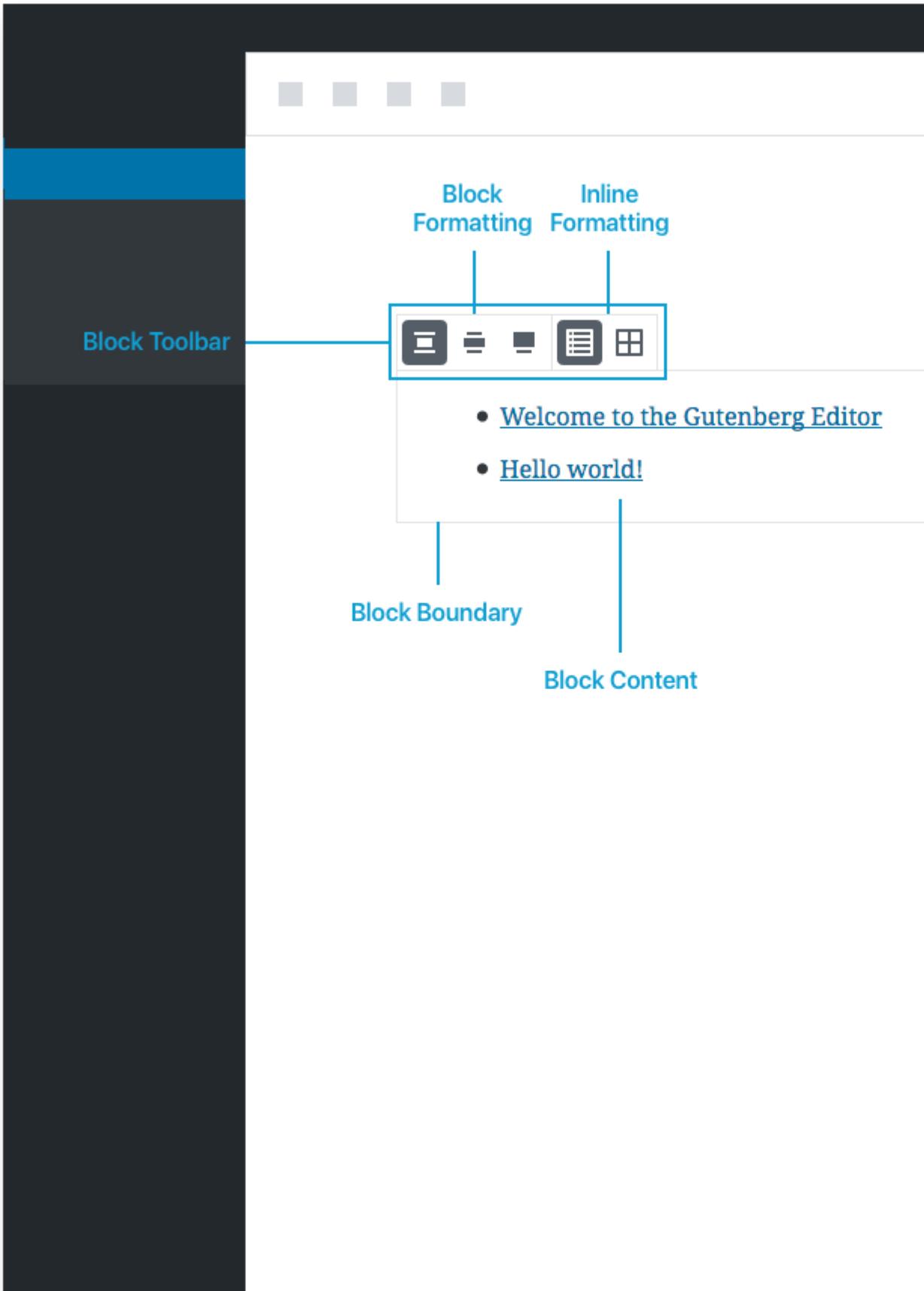
Block settings

- Has description: “They’re worth 1,000 words! Insert a single image.”
- Has options for changing or adding alt text and adding additional custom CSS classes.

Future improvements to the Image block could include getting rid of the media modal in place of letting users select images directly from the placeholder itself. In general, try to avoid modals.

[Latest Post](#)

Large Screens



Placeholder

Has no placeholder as it works immediately upon insertion. The default inserted state shows the last 5 posts.

Selected state

- Block Toolbar: Alignments
- Block Toolbar: Options for picking list view or grid view

Note that the Block Toolbar does not include the Block Chip in this case, since there are no similar blocks to switch to.

Block settings

- Has description: “Display a list of your most recent posts.”
- Has options for post order, narrowing the list by category, changing the default number of posts to show, and showing the post date.

Latest Posts is fully functional as soon as it’s inserted because it comes with good defaults.

First published

February 5, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Block Design](#)

[Previous User Interface](#) [Previous: User Interface](#)

[Next Animation](#) [Next: Animation](#)

Animation

In this article

Table of Contents

- [Principles](#)
 - [Point of Origin](#)
 - [Speed](#)
 - [Simple](#)
 - [Consistency](#)
- [Accessibility Considerations](#)
- [Inventory of Reused Animations](#)

[↑ Back to top](#)

Animation can help reinforce a sense of hierarchy and spatial orientation. This document goes into principles you should follow when you add animation.

Principles

Point of Origin

- Animation can help anchor an interface element. For example a menu can scale up from the button that opened it.
- Animation can help give a sense of place; for example a sidebar can animate in from the side, implying it was always hidden off-screen.
- Design your animations as if you’re working with real-world materials. Imagine your user interface elements are made of real materials — when not on screen, where are they? Use animation to help express that.

Speed

- Animations should never block a user interaction. They should be fast, almost always complete in less than 0.2 seconds.
- A user should not have to wait for an animation to finish before they can interact.
- Animations should be performant. Use `transform` CSS properties when you can, these render elements on the GPU, making them smooth.
- If an animation can’t be made fast & performant, leave it out.

Simple

- Don’t bounce if the material isn’t made of rubber.
- Don’t rotate, fold, or animate on a curved path. Keep it simple.

Consistency

In creating consistent animations, we have to establish physical rules for how elements behave when animated. When all animations follow these rules, they feel consistent, related, and predictable. An animation should match user expectations, if it doesn’t, it’s probably not the right animation for the job.

Reuse animations if one already exists for your task.

Accessibility Considerations

- Animations should be subtle. Be cognizant of users with [vestibular disorders triggered by motion](#).
- Don’t animate elements that are currently reporting content to adaptive technology (e.g., an `aria-live` region that’s receiving updates). This can cause confusion wherein the technology tries to parse a region that’s actively changing.
- Avoid animations that aren’t directly triggered by user behaviors.
- Whenever possible, ensure that animations respect the OS-level “Reduce Motion” settings. This can be done by utilizing the [`prefers-reduced-motion`](#) media query. Gutenberg includes a `@reduce-motion` mixin for this, to be used alongside rules that include a CSS `animate` property.

Inventory of Reused Animations

The generic `Animate` component is used to animate different parts of the interface. See [the component documentation](#) for more details about the available animations.

First published

February 5, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Animation”](#)

[Previous Block Design](#) [Previous: Block Design](#)

[Next Resources](#) [Next: Resources](#)

Resources

In this article

Table of Contents

- [Figma](#)
 - [How to contribute](#)
 - [Resources for learning how to use Figma](#)
 - [Learning how to use files and projects](#)
 - [Learning how to use components](#)
 - [Learning how to use WordPress Figma libraries](#)

[↑ Back to top](#)

Figma

The [WordPress Design team](#) uses [Figma](#) to collaborate and share work. If you'd like to contribute, join the [#design channel](#) in [Slack](#) and ask the team to set you up with a free Figma account. This will give you access to a helpful library of components used in WordPress. They are stable, fully supported, up to date, and ready for use in designs and prototypes.

How to contribute

Resources for learning how to use Figma

[Getting started with Figma](#)

[Top Online Tutorials to Learn Figma for UI/UX Design](#)

[Take a Tour Around Figma](#)

[Learning how to use files and projects](#)

[Getting started with Figma files and projects](#)

[What are files?](#)

[What are projects?](#)

[Video tutorial](#)

[FAQ](#)

[Learning how to use components](#)

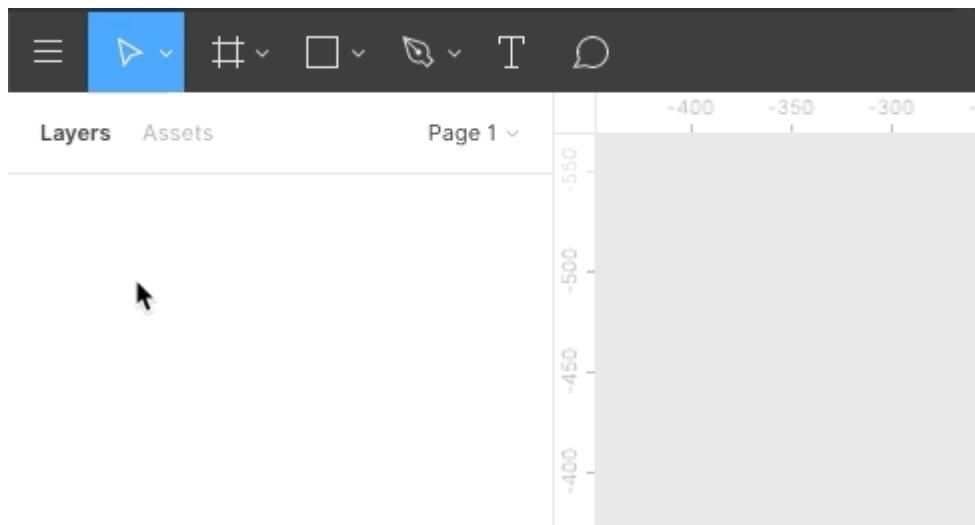
[Getting started with components](#)

[What are components?](#)

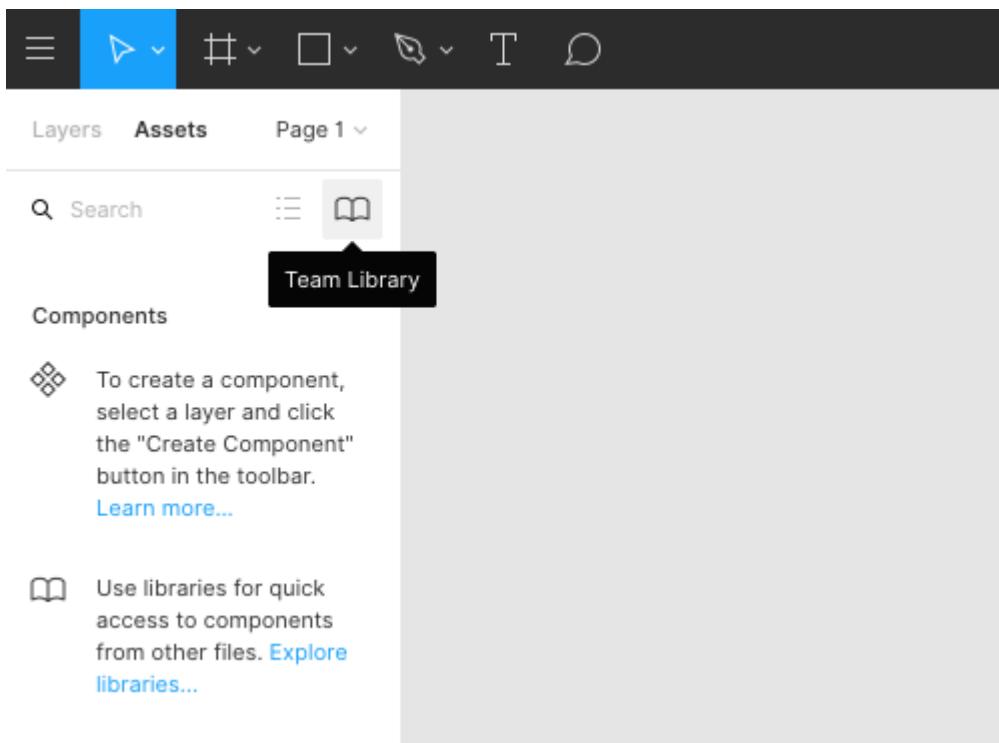
[Video tutorial](#)

[Learning how to use WordPress Figma libraries](#)

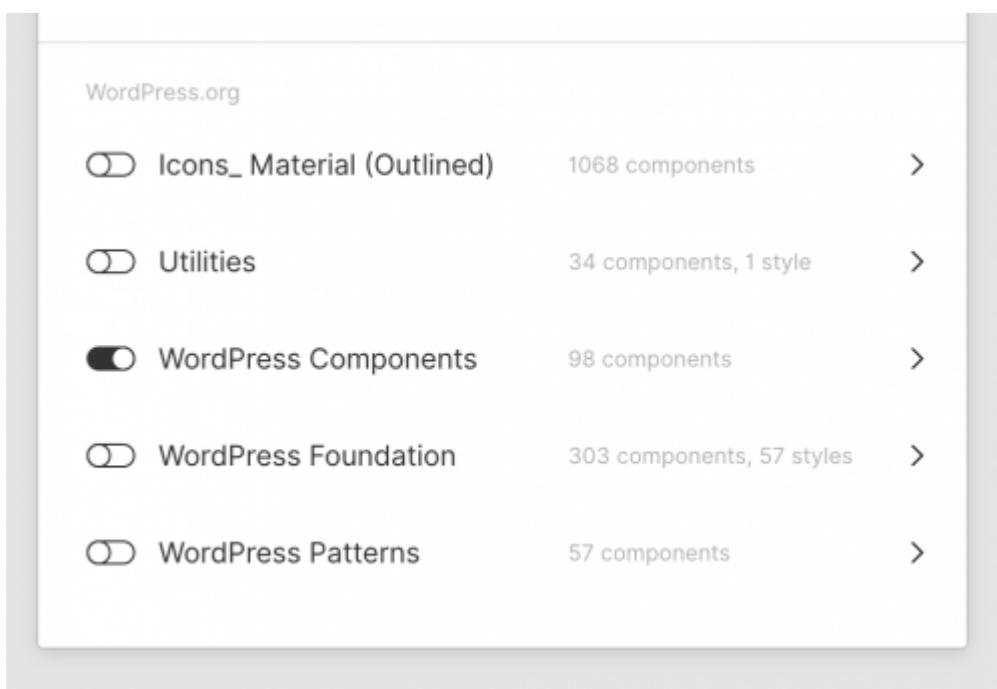
How to turn on the WordPress Components library in Figma



1. Click the **Team Library** icon in the **Assets Panel**:



1. The **Library** modal will open and allow you to view a list of available libraries. Toggle to *Enable* or *Disable* a specific library:



How to refine or contribute to the WordPress components React library (*Coming soon*)

WordPress components in Figma mirror the live React components. Documentation for how to refine or contribute to WordPress components in React is coming soon.

If you have questions, please don't hesitate to ask in the #design channel on the WordPress community Slack.

First published

February 5, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Resources”](#)

[Previous Animation](#) [Previous: Animation](#)

[Next History](#) [Next: History](#)

History

In this article

[Table of Contents](#)

- [Inspiration](#)
 - [Gutenberg updates and feature overviews](#)

[↑ Back to top](#)

A set of links and resources covering the history of the Gutenberg project, how it got started, sources of inspiration, and initial thinking.

[Inspiration](#)

A community [Editor Experience Survey](#) was conducted in early 2017, which reinforced the need for a new editing experience in WordPress.

This is a list of historical articles and products that influenced the Gutenberg project and the creation of the Block Editor.

- [Parrot: an integrated site builder and editor concept for WordPress](#)
- LivingDocs
- Apple Keynote
- Slack
- Google Sites v2

[Gutenberg updates and feature overviews](#)

- [Themes of the Future](#), Eileen Violini (January 2021)
- [Status Check: Site Editing & Customization](#), Matías Ventura Bausero (December 2020)
- [State of the Word 2020 FSE Demo](#), Matt Mullenweg (December 2020)
- [Embrace the Modularity](#), Riad Benguella (January 2020)
- [State of the Word 2019 Gutenberg Demo](#), Matt Mullenweg (December 2019)
- [Growing JavaScript Skills with WordPress](#) at JavaScript for WordPress conference (July 2019)
- [Beyond Gutenberg](#), Matías Ventura Bausero (July 2018)

- [Anatomy of a block: Gutenberg design patterns](#), Tammie Lister (July 2018)
- [The Language of Gutenberg](#), Miguel Fonseca (April 2018)
- [State of the Word 2017 Gutenberg Demo](#), Matt Mullenweg with demo by Matías Ventura Bausero (December 2017)
- [Gutenberg, or the Ship of Theseus](#), Matías Ventura Bausero (October 2017)
- [We Called It Gutenberg for a Reason](#), Matt Mullenweg (August 2017)
- [How Gutenberg is Changing WordPress Development](#), Riad Benguella (October 2017)
- [Revised suggested roadmap for Gutenberg and Customization](#), Tammie Lister (August 2017)
- [Discovering Gutenberg and next steps](#), Tammie Lister (August 2017)
- [How Gutenberg Will Shape the Future of WordPress](#), Morten Rand-Henrikson (August 2017)

You can also view this [Index of Gutenberg related posts](#) for more resources.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: History”](#)

[Previous Resources](#) [Previous: Resources](#)

[Next Contributor Guide](#) [Next: Contributor Guide](#)

Contributor Guide

In this article

[Table of Contents](#)

- [Sections](#)
 - [Repository management](#)
- [Guidelines](#)

[↑ Back to top](#)

Welcome to the Gutenberg Project Contributor Guide. This guide is here to help you get setup and start contributing to the project. If you have any questions, you'll find us in the #core-editor channel in the WordPress Core Slack, [free to join](#).

Gutenberg is a sub-project of Core WordPress. Please see the [Core Contributor Handbook](#) for additional information.

Sections

Find the section below based on what you are looking to contribute:

- **Code?** See the [developer section](#).
- **Design?** See the [design section](#).
- **Documentation?** See the [documentation section](#)
- **Triage Support?** See the [triaging issues section](#)
- **Internationalization?** See the [localizing and translating section](#)

Repository management

The Gutenberg project uses GitHub for managing code and tracking issues. Please see the following sections for the project methodologies using GitHub.

- [Issue Management](#)
- [Pull Requests](#)
- [Teams and Projects](#)

Guidelines

See the [Contributing Guidelines](#) for the rules around contributing: This includes the code of conduct and licensing information.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Contributor Guide”](#)

[Previous History](#) [Previous: History](#)

[Next Code Contributions](#) [Next: Code Contributions](#)

Code Contributions

In this article

Table of Contents

- [Discussions](#)

- [Development Hub](#)
- [Contributor resources](#)

[↑ Back to top](#)

A guide on how to get started contributing code to the Gutenberg project.

Discussions

The [Make WordPress Core blog](#) is the primary spot for the latest information around WordPress development: including announcements, product goals, meeting notes, meeting agendas, and more.

Real-time discussions for development take place in `#core-editor` and `#core-js` channels in [Make WordPress Slack](#) (registration required). Weekly meetings for the editor component are on Wednesdays at 14:00UTC, and for the JavaScript component on Tuesday at 15:00UTC, in their respective Slack channels.

Development Hub

The Gutenberg project uses GitHub for managing code and tracking issues. The main repository is at: <https://github.com/WordPress/gutenberg>.

Browse [the issues list](#) to find issues to work on. The [good first issue](#) and [good first review](#) labels are good starting points.

Contributor resources

- [Getting Started](#) documents getting your development environment setup, this includes your test site and developer tools suggestions.
- [Git Workflow](#) documents the git process for deploying changes using pull requests.
- [Coding Guidelines](#) outline additional patterns and conventions used in the Gutenberg project.
- [Testing Overview](#) for PHP and JavaScript development in Gutenberg.
- [Accessibility Testing](#) documents the process of testing accessibility in Gutenberg.
- [Managing Packages](#) documents the process for managing the npm packages.
- [Gutenberg Release Process](#) – a checklist for the different types of releases for the Gutenberg project.
- [React Native mobile editor](#) – a guide on contributing to the React Native mobile editor.
- [React Native Integration Test Guide](#) – a guide on creating integration tests for the mobile editor.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

Getting Started With Code Contribution

In this article

[Table of Contents](#)

- [Prerequisites](#)
- [Getting the Gutenberg code](#)
- [Building Gutenberg as a plugin](#)
- [Local WordPress Environment](#)
 - [Using Docker and wp-env](#)
 - [Using Local or MAMP](#)
 - [On a remote server](#)
- [Storybook](#)
- [Developer tools](#)
 - [EditorConfig](#)
 - [ESLint](#)
 - [Prettier](#)
 - [TypeScript](#)

[↑ Back to top](#)

The following guide is for setting up your local environment to contribute to the Gutenberg project. There is significant overlap between an environment to contribute and an environment used to extend the WordPress block editor. You can review the [Development Environment tutorial](#) for additional setup information.

[Prerequisites](#)

- Node.js

Gutenberg is a JavaScript project that requires [Node.js](#). The project is currently built using Node.js v20 and npm v10. Though best efforts are made to always use the Active LTS version of Node.js, this will not always be the case. For more details, please refer to the [Node.js release schedule](#).

We recommend using the [Node Version Manager](#) (nvm) since it is the easiest way to install and manage node for macOS, Linux, and Windows 10 using WSL2. See [our Development Tools guide](#) or the Nodejs site for additional installation instructions.

- Git

Gutenberg is using git for source control. Make sure you have an updated version of git installed on your computer, as well as a GitHub account. You can read the [Git Workflow](#) to learn more about using git and GitHub with Gutenberg

- [Recommended] Docker Desktop

We recommend using the [wp-env package](#) for setting WordPress environment locally.

You'll need to install Docker to use `wp - env`. See the [Development Environment tutorial for additional details](#).

Note: To install Docker on Windows 10 Home Edition, follow the [install instructions from Docker for Windows with WSL2](#).

As an alternative to Docker setup, you can use [Local](#), [WampServer](#), or [MAMP](#), or even use a remote server.

- GitHub CLI

Although not a requirement, the [GitHub CLI](#) can be very useful in helping you checkout pull requests locally. Both from the Gutenberg repo and forked repos. This can be a major time saver while code reviewing and testing pull requests.

[Getting the Gutenberg code](#)

Fork the Gutenberg repository, clone it to your computer and add the WordPress repository as upstream.

```
$ git clone https://github.com/YOUR_GITHUB_USERNAME/gutenberg.git  
$ cd gutenberg  
$ git remote add upstream https://github.com/WordPress/gutenberg.git
```

[Building Gutenberg as a plugin](#)

Install the Gutenberg dependencies and build your code in development mode:

```
npm install  
npm run dev
```

Note: The install scripts require [Python](#) to be installed and in the path of the local system. This might be installed by default for your operating system, or require downloading and installing.

There are two ways to build your code. While developing, you probably will want to use `npm run dev` to run continuous builds automatically as source files change. The dev build also includes additional warnings and errors to help troubleshoot while developing. Once you are happy with your changes, you can run `npm run build` to create optimized production build.

Once built, Gutenberg is ready to be used as a WordPress plugin!

[Local WordPress Environment](#)

To test a WordPress plugin, you need to have WordPress itself installed. If you already have a WordPress environment setup, use the above Gutenberg build as a standard WordPress plugin by putting the `gutenberg` directory in your `wp-content/plugins/` directory.

If you do not have a local WordPress environment setup, follow the steps in the rest of this section to create one.

Using Docker and wp-env

The [wp-env package](#) was developed with the Gutenberg project as a quick way to create a standard WordPress environment using Docker. It is also published as the `@wordpress/env` npm package.

By default, `wp-env` can run in a plugin directory to create and run a WordPress environment, mounting and activating the plugin automatically. You can also configure `wp-env` to use existing installs, multiple plugins, or themes. See the [wp-env package](#) for complete documentation.

Make sure Docker is running, and start `wp-env` from within the `gutenberg` directory:

```
npm run wp-env start
```

This script will create a Docker instance behind the scenes with the latest WordPress Docker image, and then will map the Gutenberg plugin code from your local copy to the environment as a Docker volume. This way, any changes you make to the code locally are reflected immediately in the WordPress instance.

Note: `npm run` will use the `wp-env / WordPress??` version specified within the Gutenberg project, making sure you are running the latest `wp-env` version.

To stop the running environment:

```
npm run wp-env stop
```

If everything went well, you should see the following message in your terminal:

```
WordPress development site started at http://localhost:8888/  
WordPress test site started at http://localhost:8889/  
MySQL is listening on port 51220
```

✓ Done! (in 261s 898ms)

And if you open Docker dashboard by rightclicking the icon in the menu bar(on Mac) or system tray (on Linux and Windows) and selecting ‘Dashboard’, you will see that the script has downloaded some Docker Images, and is running a Docker Container with fully functional WordPress installation:



Containers / Apps

Images



Search...



d2826876

RUNNING



d282

RUNN



d282

RUNN



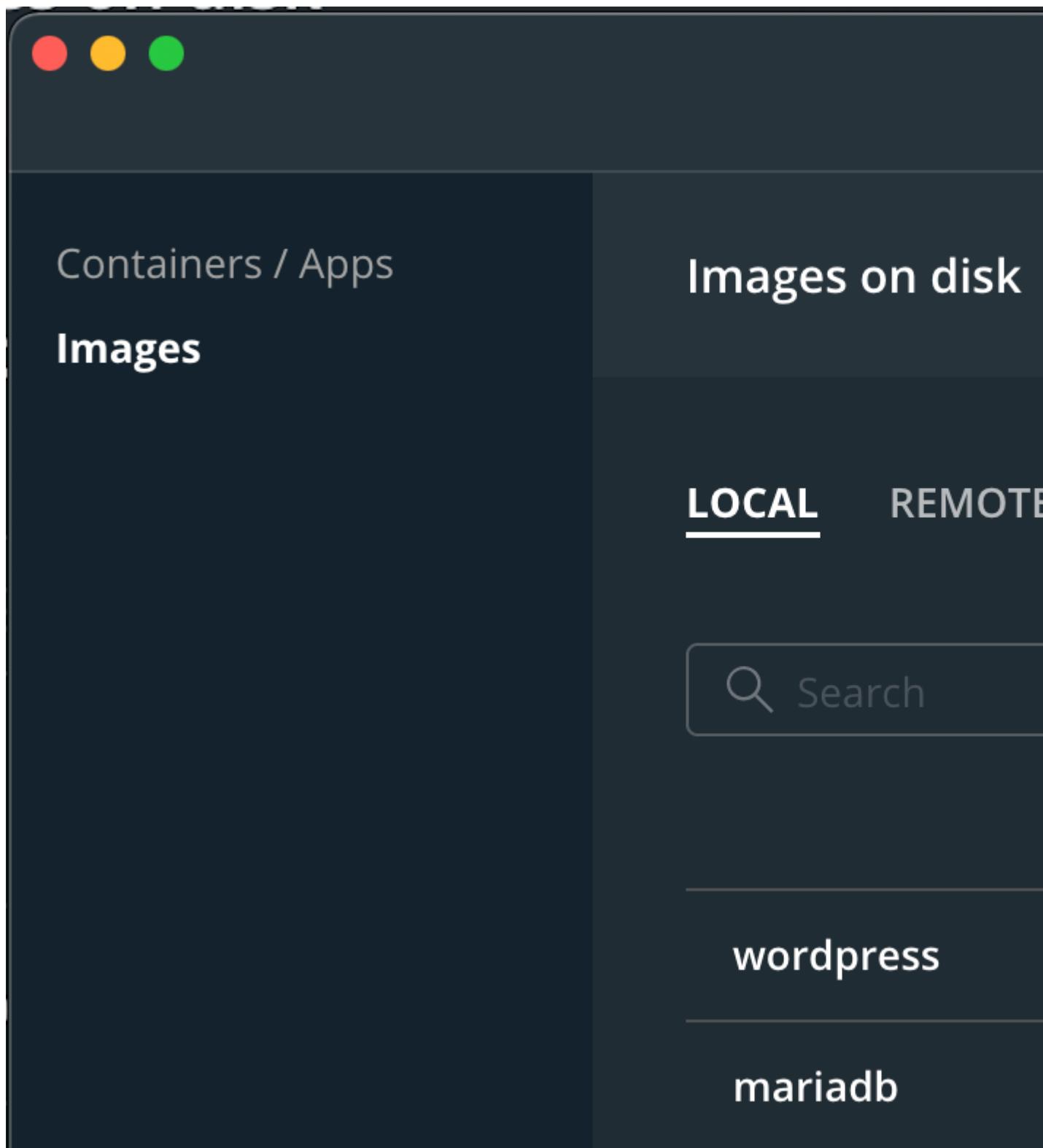
d282

RUNN



d282

RUNN



To destroy the install completely:

```
npm run wp-env destroy
```

Explore the [package documentation](#) for additional commands.

Accessing the Local WordPress Install

The WordPress installation should now be available at `http://localhost:8888`

You can access the Dashboard at: `http://localhost:8888/wp-admin/`` using `**Username**:admin`, `**Password**:password`. You'll notice the Gutenberg plugin installed and activated, this is your local build.

Accessing the MySQL Database

To access the MySQL database on the `wp-env` instance you will first need the connection details. To do this:

1. In a terminal, navigate to your local Gutenberg repo.
2. Run `npm run wp-env start` – various information about the `wp-env` environment should be logged into the terminal.
3. In the output from step 2, look for information about the *MySQL* port:
For example:

MySQL is listening on port {MySQL_Port_Number}

1. Copy / make a note of this port number (note this will change each time `wp-env` restarts).
2. You can now connect to the MySQL instance using the following details (being sure to replace {MySQL_Port_Number} with the port number from step three):

Host: 127.0.0.1
Username: root
Password: password
Database: wordpress
Port: {MySQL_Port_Number}

Please note: the MySQL port number will change each time `wp-env` restarts. If you find you can no longer access your database, simply repeat the steps above to find the new port number and restore your connection.

Tip: [Sequel Ace](#) is a useful GUI tool for accessing a MySQL database. Other tools are available and documented in this [article on accessing the WordPress database](#).

Troubleshooting

If you run into an issue, check the [troubleshooting section in wp-env documentation](#).

Using Local or MAMP

As an alternative to Docker and `wp-env`, you can also use [Local](#), [WampServer](#), or [MAMP](#) to run a local WordPress environment. To do so clone and install Gutenberg as a regular plugin in your installation by creating a symlink or copying the directory to the proper `wp-content/plugins` directory.

You will also need some extra configuration to be able to run the e2e tests.

Change the current directory to the `plugins` folder and symlink all e2e test plugins:

```
ln -s gutenberg/packages/e2e-tests/plugins/* .
```

You'll need to run this again if new plugins are added. To run e2e tests:

```
WP_BASE_URL=http://localhost:8888/gutenberg/ npm run test:e2e
```

Caching of PHP files

You'll need to disable OPCache in order to correctly work on PHP files. To fix:

- Go to **MAMP > Preferences > PHP**
- Under **Cache**, select **off**
- Confirm with **OK**

Incoming connections

By default, the web server (Apache) launched by MAMP will listen to all incoming connections, not just local ones. This means that anyone on the same local network (and, in certain cases, anyone on the Internet) can access your web server. This may be intentional and useful for testing sites on other devices, but most often this can be a privacy or security issue. Keep this in mind and don't store sensitive information in this server.

While it is possible to fix this, you should fix it at your own risk, since it breaks MAMP's ability to parse web server configurations and, as a result, makes MAMP think that Apache is listening to the wrong port. Consider switching away from MAMP. Otherwise, you can use the following:

- Edit `/Applications/MAMP/conf/apache/httpd.conf`
- Change `Listen 8888` to `Listen 127.0.0.1:8888`

Linking to other directories

You may like to create links in your `plugins` and `themes` directories to other folders, e.g.

- `wp-content/plugins/gutenberg -> ~/projects/gutenberg`
- `wp-content/themes/twentytwenty -> ~/projects/twentytwenty`

If so, you need to instruct Apache to allow following such links:

- Open or start a new file at `/Applications/MAMP/htdocs/.htaccess`
- Add the following line: `Options +SymLinksIfOwnerMatch`

Using WP-CLI

Tools like MAMP tend to configure MySQL to use ports other than the default 3306, often preferring 8889. This may throw off WP-CLI, which will fail after trying to connect to the database. To remedy this, edit `wp-config.php` and change the `DB_HOST` constant from `define('DB_HOST', 'localhost')` to `define('DB_HOST', '127.0.0.1:8889').`

On a remote server

You can use a remote server in development by building locally and then uploading the built files as a plugin to the remote server.

To build: open a terminal (or if on Windows, a command prompt) and navigate to the repository you cloned. Now type `npm ci` to get the dependencies all set up. Once that finishes, you can type `npm run build`.

After building the cloned gutenberg directory contains the complete plugin, you can upload the entire repository to your `wp-content/plugins` directory and activate the plugin from the WordPress admin.

Another way to upload after building is to run `npm run build:plugin-zip` to create a plugin zip file — this requires `bash` and `php` to run. The script creates `gutenberg.zip` that you can use to install Gutenberg through the WordPress admin.

[Storybook](#)

Storybook is an open source tool for developing UI components in isolation for React, React Native and more. It makes building stunning UIs organized and efficient.

The Gutenberg repository also includes [Storybook](#) integration that allows testing and developing in a WordPress-agnostic context. This is very helpful for developing reusable components and trying generic JavaScript modules without any backend dependency.

You can launch Storybook by running `npm run storybook:dev` locally. It will open in your browser automatically.

You can also test Storybook for the current `trunk` branch on GitHub Pages: <https://wordpress.github.io/gutenberg/>

[Developer tools](#)

We recommend configuring your editor to automatically check for syntax and lint errors. This will help you save time as you develop by automatically fixing minor formatting issues. Here are some directions for setting up Visual Studio Code, a popular editor used by many of the core developers, these tools are also available for other editors.

[EditorConfig](#)

[EditorConfig](#) defines a standard configuration for setting up your editor, for example using tabs instead of spaces. You should install the [EditorConfig for VS Code](#) extension and it will automatically configure your editor to match the rules defined in [.editorconfig](#).

[ESLint](#)

[ESLint](#) statically analyzes the code to find problems. The lint rules are integrated in the continuous integration process and must pass to be able to commit. You should install the [ESLint Extension](#) for Visual Studio Code, see eslint docs for [more editor integrations](#).

With the extension installed, ESLint will use the `.eslintrc.js` file in the root of the Gutenberg repository for formatting rules. It will highlight issues as you develop, you can also set the following preference to fix lint rules on save.

```
"editor.codeActionsOnSave": {  
    "source.fixAll.eslint": true  
},
```

[Prettier](#)

[Prettier](#) is a tool that allows you to define an opinionated format, and automate fixing the code to match that format. Prettier and ESLint are similar, Prettier is more about formatting and style, while ESLint is for detecting coding errors.

To use Prettier with Visual Studio Code, you should install the [Prettier – Code formatter extension](#). You can then configure it to be the default formatter and to automatically fix issues on save, by adding the following to your settings. **Note: depending on where you are viewing this document, the brackets may show as double, the proper format is just a single bracket.**

```
"[ [javascript]]": {  
    "editor.defaultFormatter": "esbenp.prettier-vscode",  
    "editor.formatOnSave": true  
},  
"[ [markdown]]": {  
    "editor.defaultFormatter": "esbenp.prettier-vscode",  
    "editor.formatOnSave": true  
},
```

This will use the `.prettierrc.js` file included in the root of the Gutenberg repository. The config is included from the [@wordpress/prettier-config](#) package.

If you only want to use this configuration with the Gutenberg project, create a directory called `.vscode` at the top-level of Gutenberg, and place your settings in a `settings.json` there. Visual Studio Code refers to this as Workplace Settings, and only apply to the project.

For other editors, see [Prettier's Editor Integration docs](#)

[TypeScript](#)

TypeScript is a typed superset of JavaScript language. The Gutenberg project uses TypeScript via JSDoc to [type check JavaScript files](#). If you use Visual Studio Code, TypeScript support is built-in, otherwise see [TypeScript Editor Support](#) for editor integrations.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Getting Started With Code Contribution”](#)

[Previous Code Contributions](#) [Previous: Code Contributions](#)

[Next Git Workflow](#) [Next: Git Workflow](#)

Git Workflow

In this article

Table of Contents

- [Overview](#)
- [Git Workflow Walkthrough](#)
- [Branch naming](#)
- [Keeping your branch up to date](#)
- [Keeping your fork up to date](#)
- [Miscellaneous](#)
 - [Git archeology](#)

[↑ Back to top](#)

This documentation is intended to help you get started using git with Gutenberg. Git is a powerful source code management tool; to learn git deeply, check out the [Pro Git book](#) available free online under CC BY-NC-SA 3.0 license.

If you are unfamiliar with using git, it is worthwhile to explore and play with it. Try out the [git tutorial](#) as well as the [git user manual](#) for help getting started.

The Gutenberg project follows a standard pull request process for contributions. See GitHub's documentation for [additional details about pull requests](#).

[Overview](#)

An overview of the process for contributors is:

- Fork the Gutenberg repository.
- Clone the forked repository.
- Create a new branch.
- Make code changes.
- Confirm tests pass.
- Commit the code changes within the newly created branch.
- Push the branch to the forked repository.
- Submit a pull request to the Gutenberg repository.

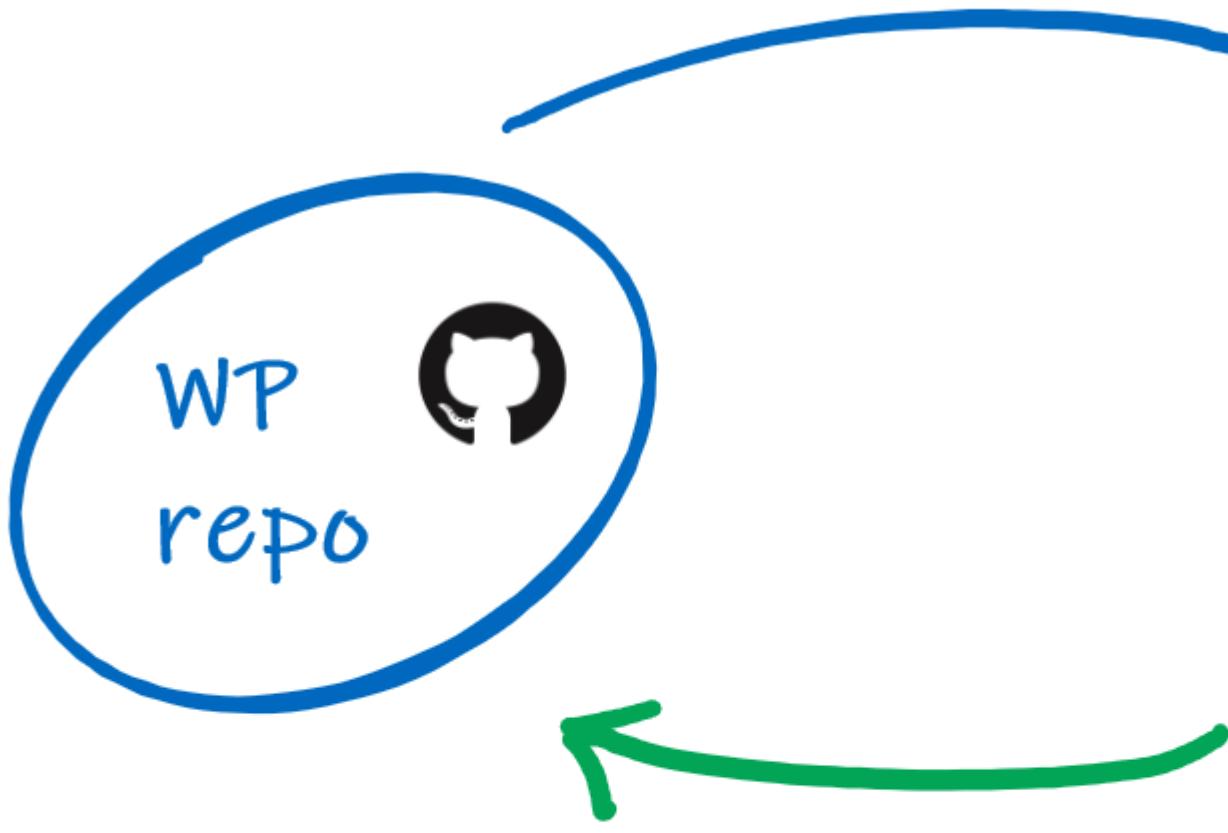
See the [repository management document](#) for additional information on how the Gutenberg project uses GitHub.

[Git Workflow Walkthrough](#)

The workflow for code and documentation is the same, since both are managed in GitHub. You can watch a [video walk-through of contributing documentation](#) and the accompanying [tutorial for contributing to Gutenberg](#).

Here is a visual overview of the Git workflow:

1. fork



Step 1: Go to the Gutenberg repository on GitHub and click Fork. This creates a copy of the main Gutenberg repository to your account.



WordPress/gutenberg: Th x



github.com/WordPress/gutenberg



Search or jump to...



R



WordPress / **gutenberg**

<> Code

! Issues

2.4k

Pull requ



master ▾



659 branches



2,596 ta



Addison-Stavlo Fix Template Part auto-draf



.github

Update code o



bin

Fix error handl



docs

Fix getEntityRe



lib

Fix Template F

Step 2: Clone your forked repository locally. It is located at: 'https://github.com/YOUR-USER-NAME/gutenberg'. Cloning copies all the files to your computer. Open a terminal and run:

```
git clone https://github.com/YOUR-USER-NAME/gutenberg
```

This will create a directory called `gutenberg` with all the files for the project. It might take a couple of minutes because it is downloading the entire history of the Gutenberg project.

Step 3: Create a branch for your change (see below for branch naming). For this example, the branch name is the complete string: `update/my-branch`

```
git switch -c update/my-branch
```

Step 4: Make the code changes. Build, confirm, and test your change thoroughly. See [coding guidelines](#) and [testing overview](#) for guidance.

Step 5: Commit your change with a [good commit message](#). This will commit your change to your local copy of the repository.

```
git commit -m "Your Good Commit Message" path/to/FILE
```

Step 6: Push your change up to GitHub. The change will be pushed to your fork of the repository on the GitHub

```
git push -u origin update/my-branch
```

Step 7: Go to your forked repository on GitHub — it will automatically detect the change and give you a link to create a pull request.



maruskaz/gutenberg: Th x



github.com/maruskaz/gutenberg



Search or jump to...



R

maruskaz / **gutenberg**

forked from WordPress/gutenberg

<> **Code**

Pull requests

Actions

master ▾

655 branches

2,531 ta

This branch is 1 commit ahead of WordPress:master



mkaz TEST



.github

Update code



bin

Fix error handl

Step 8: Create the pull request. This will create the request on the WordPress Gutenberg repository to integrate the change from your forked repository.

Step 9: Keep up with new activity on the pull request. If any additional changes or updates are requested, then make the changes locally and push them up, following Steps 4-6.

Do not make a new pull request for updates; by pushing your change to your repository it will update the same PR. In this sense, the PR is a pointer on the WordPress Gutenberg repository to your copy. So when you update your copy, the PR is also updated.

That's it! Once approved and merged, your change will be incorporated into the main repository.


Branch naming

You should name your branches using a prefixes and short description, like this: [type] / [change].

Suggested prefixes:

- `add/` = add a new feature
- `try/` = experimental feature, “tentatively add”
- `update/` = update an existing feature
- `remove/` = remove an existing feature
- `fix/` = fix an existing issue

For example, `add/gallery-block` means you're working on adding a new gallery block.

Keeping your branch up to date

When many different people are working on a project simultaneously, pull requests can go stale quickly. A “stale” pull request is one that is no longer up to date with the main line of development, and it needs to be updated before it can be merged into the project.

There are two ways to do this: merging and rebasing. In Gutenberg, the recommendation is to rebase. Rebasing means rewriting your changes as if they're happening on top of the main line of development. This ensures the commit history is always clean and linear. Rebasing can be performed as many times as needed while you're working on a pull request. **Do share your work early on** by opening a pull request and keeping your history rebase as you progress.

The main line of development is known as the `trunk` branch. If you have a pull-request branch that cannot be merged into `trunk` due to a conflict (this can happen for long-running pull requests), then in the course of rebasing you'll have to manually resolve any conflicts in your local copy. Learn more in [section *Perform a rebase*](#) of *How to Rebase a Pull Request*.

Once you have resolved any conflicts locally you can update the pull request with `git push --force-with-lease`. Using the `--force-with-lease` parameter is important to guarantee that you don't accidentally overwrite someone else's work.

To sum it up, you need to fetch any new changes in the repository, rebase your branch on top of `trunk`, and push the result back to the repository. These are the corresponding commands:

```
git fetch  
git rebase trunk  
git push --force-with-lease origin your-branch-name
```

Keeping your fork up to date

Working on pull request starts with forking the Gutenberg repository, your separate working copy. Which can easily go out of sync as new pull requests are merged into the main repository. Here your working repository is a **fork** and the main Gutenberg repository is **upstream**. When working on new pull request you should always update your fork before you do `git checkout -b my-new-branch` to work on a feature or fix.

You will need to add an `upstream` remote in order to keep your fork updated.

```
git remote add upstream https://github.com/WordPress/gutenberg.git  
git remote -v  
origin git@github.com:your-account/gutenberg.git (fetch)  
origin git@github.com:your-account/gutenberg.git (push)  
upstream https://github.com/WordPress/gutenberg.git (fetch)  
upstream https://github.com/WordPress/gutenberg.git (push)
```

To sync your fork, you first need to fetch the upstream changes and merge them into your local copy:

```
git fetch upstream  
git checkout trunk  
git merge upstream/trunk
```

Once your local copy is updated, push your changes to update your fork on GitHub:

```
git push
```

The above commands will update your `trunk` branch from *upstream*. To update any other branch replace `trunk` with the respective branch name.

Miscellaneous

Git archeology

When looking for a commit that introduced a specific change, it might be helpful to ignore revisions that only contain styling or formatting changes.

Fortunately, newer versions of `git` gained the ability to skip commits in history:

```
git blame --ignore-rev f63053cace3c02e284f00918e1854284c85b9132 -L 66,73 p
```

All styling and formatting revisions are tracked using the `.git-blame-ignore-revs` file in the Gutenberg repository. You can use this file to ignore them all at once:

```
git blame --ignore-revs-file .git-blame-ignore-revs -L 66,73 packages/api-
```

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Git Workflow”](#)

[Previous](#) [Getting Started With Code Contribution](#) [Previous: Getting Started With Code Contribution](#)
[Next Coding Guidelines](#) [Next: Coding Guidelines](#)

Coding Guidelines

In this article

Table of Contents

- [CSS](#)
 - [Naming](#)
- [JavaScript](#)
 - [Imports](#)
 - [Legacy experimental APIs, plugin-only APIs, and private APIs](#)
 - [Exposing private APIs publicly](#)
 - [Objects](#)
 - [Strings](#)
 - [Optional chaining](#)
 - [React components](#)
- [JavaScript documentation using JSDoc](#)
 - [Custom types](#)
 - [Importing and exporting types](#)
 - [Generic types](#)
 - [Nullable, undefined, and void types](#)
 - [Documenting examples](#)
 - [Documenting React components](#)
- [PHP](#)

[↑ Back to top](#)

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 descendants.

Example:

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

```
export default function Notice( { children, onRemove } ) {  
  return (  
    <div className="components-notice">  
      <div className="components-notice__content">{ children }</div>  
      <Button  
        className="components-notice__dismiss"  
        icon={ check }  
        label={ __( 'Dismiss this notice' ) }  
        onClick={ onRemove }  
      />  
    </div>  
  );  
}
```

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 again the Notices example. We may want to apply specific styling for dismissible notices. The [classnames package](#) can be a helpful utility for conditionally applying modifier class names.

```
import classnames from 'classnames';  
  
export default function Notice( { children, onRemove, isDismissible } ) {  
  const classes = classnames( 'components-notice', {  
    'is-dismissable': isDismissible,  
  } );  
  
  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](#)

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.

These separations are identified by multi-line comments at the top of a file which imports code from another file or source.

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:

```
/**  
 * External dependencies  
 */  
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:

```
/**  
 * WordPress dependencies  
 */  
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:

```
/**  
 * Internal dependencies  
 */  
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 process.env.IS_GUTENBERG_PLUGIN allows Webpack to exclude this
// export from WordPress core:
if ( process.env.IS_GUTENBERG_PLUGIN ) {
    export { doSomethingExciting } from './api';
}
```

The public interface of such APIs is not yet finalized. Aside from references within the code, they 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 a 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 know using unstable features means my theme or plugin will inevitably break other packages',
        '@wordpress/block-editor' // Name of the package calling __dangerousOptInToUnstableAPIsOnlyForCoreModules
    );

```

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 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  
// 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  
// 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 an private argument to a stable component you'll need to prepare a stable and an 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/actions.js](#):

```
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
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 process.env.IS_GUTENBERG_PLUGIN check.
if ( process.env.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, the [use of TypeScript tooling](#) for types validation, and automated documentation generation using [@wordpress/docgen](#).

For additional guidance, consult the following resources:

- [JSDoc Official Documentation](#)
- [TypeScript Supported JSDoc](#)

Custom types

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  
 *                                     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 recommended 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

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".

A small screenshot of a GitHub commit message showing the code above and a green checkmark indicating it works correctly.

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

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]} */
/** @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  
 * expected to call the `next` middleware once it has completed its handling.  
 *  
 * @typedef {(options:WPAPIFetchOptions,next:WPAPIFetchMiddleware)=>void}  
 */  
  
/**  
 * 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](#)

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

```

[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

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`.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Coding Guidelines”](#)

[Previous Git Workflow](#) [Previous: Git Workflow](#)
[Next Testing Overview](#) [Next: Testing Overview](#)

Testing Overview

In this article

Table of Contents

- [Why test?](#)
- [JavaScript testing](#)
 - [Folder structure](#)
 - [Importing tests](#)
 - [Describing tests](#)
 - [Setup and teardown methods](#)
 - [Mocking dependencies](#)
 - [Testing globals](#)
 - [User interactions](#)
 - [Integration testing for block UI](#)
 - [Snapshot testing](#)
 - [Debugging Jest unit tests](#)
- [Native mobile testing](#)
 - [Debugging the native mobile unit tests](#)
 - [Native mobile end-to-end tests](#)
 - [Native mobile integration tests](#)
- [End-to-end testing](#)
 - [Using wp-env](#)
 - [Using alternate environment](#)
 - [Scenario testing](#)
 - [Core block testing](#)
 - [Flaky tests](#)

- [PHP testing](#)
- [Performance testing](#)

[↑ Back to top](#)

Gutenberg contains both PHP and JavaScript code, and encourages testing and code style linting for both.

Why test?

Aside from the joy testing will bring to your life, tests are important not only because they help to ensure that our application behaves as it should, but also because they provide concise examples of how to use a piece of code.

Tests are also part of our code base, which means we apply to them the same standards we apply to all our application code.

As with all code, tests have to be maintained. Writing tests for the sake of having a test isn't the goal – rather we should try to strike the right balance between covering expected and unexpected behaviours, speedy execution and code maintenance.

When writing tests consider the following:

- What behaviour(s) are we testing?
- What errors are likely to occur when we run this code?
- Does the test test what we think it is testing? Or are we introducing false positives/negatives?
- Is it readable? Will other contributors be able to understand how our code behaves by looking at its corresponding test?

JavaScript testing

Tests for JavaScript use [Jest](#) as the test runner and its API for [globals](#) (`describe`, `test`, `beforeEach` and so on) [assertions](#), [mocks](#), [spies](#) and [mock functions](#). If needed, you can also use [React Testing Library](#) for React component testing.

It should be noted that in the past, React components were unit tested with [Enzyme](#). However, React Testing Library (RTL) is now used for all existing and new tests instead.

Assuming you've followed the [instructions](#) to install Node and project dependencies, tests can be run from the command-line with NPM:

```
npm test
```

Linting is static code analysis used to enforce coding standards and to avoid potential errors. This project uses [ESLint](#) and [TypeScript's JavaScript type-checking](#) to capture these issues. While the above `npm test` will execute both unit tests and code linting, code linting can be verified independently by running `npm run lint`. Some JavaScript issues can be fixed automatically by running `npm run lint:js:fix`.

To improve your developer workflow, you should setup an editor linting integration. See the [getting started documentation](#) for additional information.

To run unit tests only, without the linter, use `npm run test:unit` instead.

Folder structure

Keep your tests in a `test` folder in your working directory. The test file should have the same name as the test subject file.

```
+-- test
|   +-- bar.js
+-- bar.js
```

Only test files (with at least one test case) should live directly under `/test`. If you need to add external mocks or fixtures, place them in a sub folder, for example:

- `test/mocks/[file-name].js`
- `test/fixtures/[file-name].js`

Importing tests

Given the previous folder structure, try to use relative paths when importing of the **code you're testing**, as opposed to using project paths.

Good

```
import { bar } from '../bar';
```

Not so good

```
import { bar } from 'components/foo/bar';
```

It will make your life easier should you decide to relocate your code to another position in the application directory.

Describing tests

Use a `describe` block to group test cases. Each test case should ideally describe one behaviour only.

In test cases, try to describe in plain words the expected behaviour. For UI components, this might entail describing expected behaviour from a user perspective rather than explaining code internals.

Good

```
describe('CheckboxWithLabel', () => {
  test('checking checkbox should disable the form submit button', () =>
    ...
  );
});
```

Not so good

```
describe('CheckboxWithLabel', () => {
  test('checking checkbox should set this.state.disableButton to `true`', () =>
```

```
    } );
} );
```

Setup and teardown methods

The Jest API includes some nifty [setup and teardown methods](#) that allow you to perform tasks *before* and *after* each or all of your tests, or tests within a specific `describe` block.

These methods can handle asynchronous code to allow setup that you normally cannot do inline. As with [individual test cases](#), you can return a Promise and Jest will wait for it to resolve:

```
// one-time setup for *all* tests
beforeAll( () =>
  someAsyncAction().then( ( resp ) => {
    window.someGlobal = resp;
  })
);

// one-time teardown for *all* tests
afterAll( () => {
  window.someGlobal = null;
});
```

`afterEach` and `afterAll` provide a perfect (and preferred) way to ‘clean up’ after our tests, for example, by resetting state data.

Avoid placing clean up code after assertions since, if any of those tests fail, the clean up won’t take place and may cause failures in unrelated tests.

Mocking dependencies

Dependency injection

Passing dependencies to a function as arguments can often make your code simpler to test. Where possible, avoid referencing dependencies in a higher scope.

Not so good

```
import VALID_VALUES_LIST from './constants';

function isValueValid( value ) {
  return VALID_VALUES_LIST.includes( value );
}
```

Here we’d have to import and use a value from `VALID_VALUES_LIST` in order to pass:

```
expect( isValueValid( VALID_VALUES_LIST[ 0 ] ) ).toBe( true );
```

The above assertion is testing two behaviours: 1) that the function can detect an item in a list, and 2) that it can detect an item in `VALID_VALUES_LIST`.

But what if we don’t care what’s stored in `VALID_VALUES_LIST`, or if the list is fetched via an HTTP request, and we only want to test whether `isValueValid` can detect an item in a list?

Good

```
function isValidValue( value, validValuesList = [] ) {
    return validValuesList.includes( value );
}
```

Because we're passing the list as an argument, we can pass mock `validValuesList` values in our tests and, as a bonus, test a few more scenarios:

```
expect( isValidValue( 'hulk', [ 'batman', 'superman' ] ) ).toBe( false );
expect( isValidValue( 'hulk', null ) ).toBe( false );
expect( isValidValue( 'hulk', [] ) ).toBe( false );
expect( isValidValue( 'hulk', [ 'iron man', 'hulk' ] ) ).toBe( true );
```

Imported dependencies

Often our code will use methods and properties from imported external and internal libraries in multiple places, which makes passing around arguments messy and impracticable. For these cases `jest.mock` offers a neat way to stub these dependencies.

For instance, lets assume we have `config` module to control a great deal of functionality via feature flags.

```
// bilbo.js
import config from 'config';
export const isBilboVisible = () =>
    config.isEnabled( 'the-ring' ) ? false : true;
```

To test the behaviour under each condition, we stub the `config` object and use a `jest` mocking function to control the return value of `isEnabled`.

```
// test/bilbo.js
import { isEnabled } from 'config';
import { isBilboVisible } from '../bilbo';

jest.mock( 'config', () => ( {
    // bilbo is visible by default
    isEnabled: jest.fn( () => false ),
} ) );

describe( 'The bilbo module', () => {
    test( 'bilbo should be visible by default', () => {
        expect( isBilboVisible() ).toBe( true );
    } );
}

test( 'bilbo should be invisible when the `the-ring` config feature flag is enabled', () => {
    isEnabled.mockImplementationOnce( ( name ) => name === 'the-ring' );
    expect( isBilboVisible() ).toBe( false );
});
```

```
    } );
} );
```

Testing globals

We can use [Jest spies](#) to test code that calls global methods.

```
import { myModuleFunctionThatOpensANewWindow } from '../my-module';

describe( 'my module', () => {
  beforeAll( () => {
    jest.spyOn( global, 'open' ).mockImplementation( () => true );
  } );

  test( 'something', () => {
    myModuleFunctionThatOpensANewWindow();
    expect( global.open ).toHaveBeenCalled();
  } );
} );
```

User interactions

Simulating user interactions is a great way to **write tests from the user's perspective**, and therefore avoid testing implementation details.

When writing tests with Testing Library, there are two main alternatives for simulating user interactions:

1. The [fireEvent](#) API, a utility for firing DOM events part of the Testing Library core API.
2. The [user-event](#) library, a companion library to Testing Library that simulates user interactions by dispatching the events that would happen if the interaction took place in a browser.

The built-in `fireEvent` is a utility for dispatching DOM events. It dispatches exactly the events that are described in the test spec – even if those exact events never had been dispatched in a real interaction in a browser.

On the other hand, the `user-event` library exposes higher-level methods (e.g. `type`, `selectOptions`, `clear`, `doubleClick`...), that dispatch events like they would happen if a user interacted with the document, and take care of any react-specific quirks.

For the above reasons, **the user-event library is recommended when writing tests for user interactions**.

Not so good: using `fireEvent` to dispatch DOM events.

```
import { render, screen } from '@testing-library/react';

test( 'fires onChange when a new value is typed', () => {
  const spyOnChange = jest.fn();

  // A component with one `input` and one `select`.
  render( <MyComponent onChange={ spyOnChange } /> );
```

```

const input = screen.getByRole( 'textbox' );
input.focus();
// No clicks, no key events.
fireEvent.change( input, { target: { value: 62 } } );

// The `onChange` callback gets called once with '62' as the argument.
expect( spyOnChange ).toHaveBeenCalledTimes( 1 );

const select = screen.getByRole( 'listbox' );
select.focus();
// No pointer events dispatched.
fireEvent.change( select, { target: { value: 'optionValue' } } );

// ...

```

Good: using user-event to simulate user events.

```

import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

test( 'fires onChange when a new value is typed', async () => {
  const user = userEvent.setup();

  const spyOnChange = jest.fn();

  // A component with one `input` and one `select`.
  render( <MyComponent onChange={ spyOnChange } /> );

  const input = screen.getByRole( 'textbox' );
  // Focus the element, select and delete all its contents.
  await user.clear( input );
  // Click the element, type each character separately (generating keydown
  // keypress and keyup events).
  await user.type( input, '62' );

  // The `onChange` callback gets called 3 times with the following arguments:
  // - 1: clear('')
  // - 2: '6'
  // - 3: '62'
  expect( spyOnChange ).toHaveBeenCalledTimes( 3 );

  const select = screen.getByRole( 'listbox' );
  // Dispatches events for focus, pointer, mouse, click and change.
  await user.selectOptions( select, [ 'optionValue' ] );

  // ...
} );

```

Integration testing for block UI

Integration testing is defined as a type of testing where different parts are tested as a group. In this case, the parts that we want to test are the different components that are required to be rendered for a specific block or editor logic. In the end, they are very similar to unit tests as they are run

with the same command using the Jest library. The main difference is that for the integration tests the blocks are run within a [special instance of the block editor](#).

The advantage of this approach is that the bulk of a block editor's functionality (block toolbar and inspector panel interactions, etc.) can be tested without having to fire up the full e2e test framework. This means the tests can run much faster and more reliably. It is suggested that as much of a block's UI functionality as possible is covered with integration tests, with e2e tests used for interactions that require a full browser environment, eg. file uploads, drag and drop, etc.

[The Cover block](#) is an example of a block that uses this level of testing to provide coverage for a large percentage of the editor interactions.

To set up a jest file for integration tests:

```
import { initializeEditor } from 'test/integration/helpers/integration-test'

async function setup( attributes ) {
  const testBlock = { name: 'core/cover', attributes };
  return initializeEditor( testBlock );
}
```

The `initializeEditor` function returns the output of the `@testing-library/react render` method. It will also accept an array of block metadata objects, allowing you to set up the editor with multiple blocks.

The integration test editor module also exports a `selectBlock` which can be used to select the block to be tested by the aria-label on the block wrapper, eg. “Block: Cover”.

[Snapshot testing](#)

This is an overview of [snapshot testing](#) and how to best leverage snapshot tests.

TL;DR Broken snapshots

When a snapshot test fails, it just means that a component's rendering has changed. If that was unintended, then the snapshot test just prevented a bug 😊

However, if the change was intentional, follow these steps to update the snapshot. Run the following to update the snapshots:

```
# --testPathPattern is optional but will be much faster by only running many tests
npm run test:unit -- --updateSnapshot --testPathPattern path/to/tests

# Update snapshot for e2e tests
npm run test:e2e -- --updateSnapshot --testPathPattern path/to/e2e-tests

# Update snapshot for Playwright
npm run test:e2e:playwright -- --update-snapshots path/to/spec
```

1. Review the diff and ensure the changes are expected and intentional.
2. Commit.

What are snapshots?

Snapshots are just a representation of some data structure generated by tests. Snapshots are stored in files and committed alongside the tests. When the tests are run, the data structure generated is compared with the snapshot on file.

It's very easy to make a snapshot:

```
test( 'foobar test', () => {
  const foobar = { foo: 'bar' };

  expect( foobar ).toMatchSnapshot();
} );
```

This is the produced snapshot:

```
exports[ `test foobar test 1` ] = ` 
Object {
  "foo": "bar",
}
`;
```

You should never create or modify a snapshot directly, they are generated and updated by tests.

Advantages

- Trivial and concise to add tests.
- Protect against unintentional changes.
- Simple to work with.
- Reveal internal structures without running the application.

Disadvantages

- Not expressive.
- Only catch issues when changes are introduced.
- Are problematic for anything non-deterministic.

Use cases

Snapshots are mostly targeted at component testing. They make us conscious of changes to a component's structure which makes them *ideal* for refactoring. If a snapshot is kept up to date over the course of a series of commits, the snapshot diffs record the evolution of a component's structure. Pretty cool 😎

```
import { render, screen } from '@testing-library/react';
import SolarSystem from 'solar-system';

describe( 'SolarSystem', () => {
  test( 'should render', () => {
    const { container } = render( <SolarSystem /> );

    expect( container ).toMatchSnapshot();
  } );
}
```

```
test( 'should contain mars if planets is true', () => {
  const { container } = render( <SolarSystem planets /> );

  expect( container ).toMatchSnapshot();
  expect( screen.getByText( /mars/i ) ).toBeInTheDocument();
} );
}
```

Reducer tests are also a great fit for snapshots. They are often large, complex data structures that shouldn't change unexpectedly, exactly what snapshots excel at!

Working with snapshots

You might be blindsided by CI tests failing when snapshots don't match. You'll need to [update snapshots](#) if the changes are expected. The quick and dirty solution is to invoke Jest with `--updateSnapshot`. That can be done as follows:

```
npm run test:unit -- --updateSnapshot --testPathPattern path/to/tests
```

`--testPathPattern` is not required, but specifying a path will speed things up by running a subset of tests.

It's a great idea to keep `npm run test:unit:watch` running in the background as you work. Jest will run only the relevant tests for changed files, and when snapshot tests fail, just hit `u` to update a snapshot!

Pain points

Non-deterministic tests may not make consistent snapshots, so beware. When working with anything random, time-based, or otherwise non-deterministic, snapshots will be problematic.

Connected components are tricky to work with. To snapshot a connected component you'll probably want to export the unconnected component:

```
// my-component.js
export { MyComponent };
export default connect( mapStateToProps )( MyComponent );

// test/my-component.js
import { MyComponent } from '../';
// run those MyComponent tests...
```

The connected props will need to be manually provided. This is a good opportunity to audit the connected state.

Best practices

If you're starting a refactor, snapshots are quite nice, you can add them as the first commit on a branch and watch as they evolve.

Snapshots themselves don't express anything about what we expect. Snapshots are best used in conjunction with other tests that describe our expectations, like in the example above:

```

test( 'should contain mars if planets is true', () => {
  const { container } = render( <SolarSystem planets /> );

  // Snapshot will catch unintended changes
  expect( container ).toMatchSnapshot();

  // This is what we actually expect to find in our test
  expect( screen.getByText( /mars/i ) ).toBeInTheDocument();
} );

```

Another good technique is to use the `toMatchDiffSnapshot` function (provided by the [snapshot-diff package](#)), which allows to snapshot only the difference between two different states of the DOM. This approach is useful to test the effects of a prop change on the resulting DOM while generating a much smaller snapshot, like in this example:

```

test( 'should render a darker background when isShady is true', () => {
  const { container } = render( <CardBody>Body</CardBody> );
  const { container: containerShady } = render(
    <CardBody isShady>Body</CardBody>
  );
  expect( container ).toMatchSnapshot( containerShady );
} );

```

Similarly, the `toMatchStyleDiffSnapshot` function allows to snapshot only the difference between the *styles* associated to two different states of a component, like in this example:

```

test( 'should render margin', () => {
  const { container: spacer } = render( <Spacer /> );
  const { container: spacerWithMargin } = render( <Spacer margin={ 5 } /> );
  expect( spacerWithMargin ).toMatchSnapshot( spacer );
} );

```

Troubleshooting

Sometimes we need to mock refs for some stories which use them. Check the following documents to learn more:

- Why we need to use [Mocking Refs for Snapshot Testing](#) with React.

In that case, you might see test failures and `TypeError` reported by Jest in the lines which try to access a property from `ref.current`.

Debugging Jest unit tests

Running `npm run test:unit:debug` will start the tests in debug mode so a [node inspector client](#) can connect to the process and inspect the execution. Instructions for using Google Chrome or Visual Studio Code as an inspector client can be found in the [wp-scripts documentation](#).

Native mobile testing

Part of the unit-tests suite is a set of Jest tests run exercise native-mobile codepaths, developed in React Native. Since those tests run on Node, they can be launched locally on your development

machine without the need for specific native Android or iOS dev tools or SDKs. It also means that they can be debugged using typical dev tools. Read on for instructions how to debug.

Debugging the native mobile unit tests

To locally run the tests in debug mode, follow these steps:

1. Make sure you have ran `npm ci` to install all the packages
2. Run `npm run test:native:debug` inside the Gutenberg root folder, on the CLI. Node is now waiting for the debugger to connect.
3. Open `chrome://inspect` in Chrome
4. Under the “Remote Target” section, look for a `.../node_modules/.bin/jest` target and click on the “inspect” link. That will open a new window with the Chrome DevTools debugger attached to the process and stopped at the beginning of the `jest.js` file. Alternatively, if the targets are not visible, click on the `Open dedicated DevTools for Node` link in the same page.
5. You can place breakpoints or `debugger;` statements throughout the code, including the tests code, to stop and inspect
6. Click on the “Play” button to resume execution
7. Enjoy debugging the native mobile unit tests!

Native mobile end-to-end tests

Contributors to Gutenberg will note that PRs include continuous integration E2E tests running the native mobile E2E tests on Android and iOS. For troubleshooting failed tests, check our guide on [native mobile tests in continuous integration](#). More information on running these tests locally can be found in [here](#).

Native mobile integration tests

There is an ongoing effort to add integration tests to the native mobile project using the [react-native-testing-library](#) library. A guide to writing integration tests can be found [here](#).

End-to-end testing

Most existing End-to-end tests currently use [Puppeteer](#) as a headless Chromium driver to run the tests in `packages/e2e-tests`, and are otherwise still run by a [Jest](#) test runner.

There's an ongoing [project](#) to migrate them from Puppeteer to Playwright. **It's recommended to write new e2e tests in Playwright whenever possible.** The sections below mostly apply to the old Jest + Puppeteer framework. See the dedicated [guide](#) if you're writing tests with Playwright.

Using wp-env

If you're using the built-in [local environment](#), you can run the e2e tests locally using this command:

```
npm run test:e2e
```

or interactively

```
npm run test:e2e:watch
```

Sometimes it's useful to observe the browser while running tests. Then, use this command:

```
npm run test:e2e:watch -- --puppeteer-interactive
```

You can control the speed of execution with `--puppeteer-slowmo`:

```
npm run test:e2e:watch -- --puppeteer-interactive --puppeteer-slowmo=200
```

You can additionally have the devtools automatically open for interactive debugging in the browser:

```
npm run test:e2e:watch -- --puppeteer-devtools
```

[Using alternate environment](#)

If using a different setup than `wp-env`, you first need to symlink the e2e test plugins to your test site, from your site's plugins directory run:

```
ln -s gutenberg/packages/e2e-tests/plugins/* .
```

Then to run the tests, specify the base URL, username, and passwords for your site. For example, if your test site is at '`http://wp.test`', use:

```
WP_BASE_URL=http://wp.test npm run test:e2e -- --wordpress-username=admin
```

[Scenario testing](#)

If you find that end-to-end tests pass when run locally, but fail in GitHub Actions, you may be able to isolate a CPU- or network-bound race condition by simulating a slow CPU or network:

```
THROTTLE_CPU=4 npm run test:e2e
```

`THROTTLE_CPU` is a slowdown factor (in this example, a 4x slowdown multiplier)

See [Chrome docs: setCPUThrottlingRate](#)

```
SLOW_NETWORK=true npm run test:e2e
```

`SLOW_NETWORK` emulates a network speed equivalent to "Fast 3G" in the Chrome devtools.

See [Chrome docs: emulateNetworkConditions](#) and [NetworkManager.js](#)

```
OFFLINE=true npm run test:e2e
```

`OFFLINE` emulates network disconnection.

See [Chrome docs: emulateNetworkConditions](#)

[Core block testing](#)

Every core block is required to have at least one set of fixture files for its main save function and one for each deprecation. These fixtures test the parsing and serialization of the block. See [the integration tests fixtures readme](#) for more information and instructions.

Flaky tests

A test is considered to be **flaky** when it can pass and fail across multiple retry attempts without any code changes. We auto retry failed tests at most **twice** on CI to detect and report them to GitHub issues automatically under the [\[Type\] Flaky Test](#) label via [report-flaky-tests](#) GitHub action. Note that a test that failed three times in a row is not counted as a flaky test and will not be reported to an issue.

PHP testing

Tests for PHP use [PHPUnit](#) as the testing framework. If you're using the built-in [local environment](#), you can run the PHP tests locally using this command:

```
npm run test:php
```

To re-run tests automatically when files change (similar to Jest), run:

```
npm run test:php:watch
```

Note: The phpunit commands require wp-env to be running and composer dependencies to be installed. The package script will start wp-env for you if it is not already running.

In other environments, run `composer run test` and `composer run test:watch`.

Code style in PHP is enforced using [PHP_CodeSniffer](#). It is recommended that you install PHP_CodeSniffer and the [WordPress Coding Standards for PHP_CodeSniffer](#) ruleset using [Composer](#). With Composer installed, run `composer install` from the project directory to install dependencies. The above `npm run test:php` will execute both unit tests and code linting. Code linting can be verified independently by running `npm run lint:php`.

To run unit tests only, without the linter, use `npm run test:unit:php` instead.

Performance testing

To ensure that the editor stays performant as we add features, we monitor the impact pull requests and releases can have on some key metrics:

- The time it takes to load the editor.
- The time it takes for the browser to respond when typing.
- The time it takes to select a block.

Performance tests are end-to-end tests running the editor and capturing these measures. Make sure you have an e2e testing environment ready.

To set up the e2e testing environment, checkout the Gutenberg repository and switch to the branch that you would like to test. Run the following command to prepare the environment.

```
nvm use && npm install  
npm run build:packages
```

To run the tests run the following command:

```
npm run test:performance
```

This gives you the result for the current branch/code on the running environment.

In addition to that, you can also compare the metrics across branches (or tags or commits) by running the following command `./bin/plugin/cli.js perf [branches]`, example:

```
./bin/plugin/cli.js perf trunk v8.1.0 v8.0.0
```

Finally, you can pass an additional `--tests-branch` argument to specify which branch's performance test files you'd like to run. This is particularly useful when modifying/extending the perf tests:

```
./bin/plugin/cli.js perf trunk v8.1.0 v8.0.0 --tests-branch add/perf-tests
```

Note This command needs may take some time to perform the benchmark. While running make sure to avoid using your computer or have a lot of background process to minimize external factors that can impact the results across branches.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Testing Overview”](#)

[Previous Coding Guidelines](#) [Previous: Coding Guidelines](#)

[Next End-to-End Testing](#) [Next: End-to-End Testing](#)

End-to-End Testing

In this article

Table of Contents

- [Running tests](#)
- [Best practices](#)
 - [Forbid \\$, use locator instead](#)
 - [Use accessible selectors](#)
 - [Selectors are strict by default](#)
 - [Don’t overload test-utils, inline simple utils](#)
 - [Favor Page Object Model over utils](#)
 - [Restify actions to clear or set states](#)
 - [Avoid global variables](#)
 - [Make explicit assertions](#)
- [Common pitfalls](#)
 - [Overusing snapshots](#)
- [Cross-browser testing](#)

[↑ Back to top](#)

This living document serves to prescribe instructions and best practices for writing end-to-end (E2E) tests with Playwright in the Gutenberg project.

See the dedicated guide if you're working with the previous Jest + Puppeteer framework. See the [migration guide](#) if you're migrating tests from Jest + Puppeteer.

Running tests

```
# Run all available tests.  
npm run test:e2e:playwright  
  
# Run in headed mode.  
npm run test:e2e:playwright -- --headed  
  
# Run tests with specific browsers (`chromium`, `firefox`, or `webkit`).  
npm run test:e2e:playwright -- --project=webkit --project=firefox  
  
# Run a single test file.  
npm run test:e2e:playwright -- <path_to_test_file> # E.g., npm run test:e2e:playwright -- ./src/test/e2e/test.js  
  
# Debugging.  
npm run test:e2e:playwright -- --debug
```

If you're developing in Linux, it currently requires testing Webkit browsers in headed mode. If you don't want to or can't run it with the GUI (e.g. if you don't have a graphic interface), prepend the command with [xvfb-run](#) to run it in a virtual environment.

```
# Run all available tests.  
xvfb-run npm run test:e2e:playwright  
  
# Only run webkit tests.  
xvfb-run -- npm run test:e2e:playwright -- --project=webkit
```

Best practices

Read the [best practices](#) guide for Playwright.

Forbid \$, use locator instead

In fact, any API that returns `ElementHandle` is [discouraged](#). This includes `$`, `$$`, `$eval`, `$$eval`, etc. [Locator](#) is a much better API and can be used with playwright's [assertions](#). This also works great with Page Object Model since that locator is lazy and doesn't return a promise.

Use accessible selectors

Use [getByRole](#) to construct the query wherever possible. It enables us to write accessible queries without having to rely on internal implementations.

```
// Select a button which includes the accessible name "Hello World" (case-insensitive).  
page.getByRole( 'button', { name: 'Hello World' } );
```

It can also be chained to perform complex queries:

```
// Select an option with a name "Buttons" under the "Block Library" region
page.getByRole( 'region', { name: 'Block Library' } )
  .getByRole( 'option', { name: 'Buttons' } )
```

See the [official documentation](#) for more info on how to use them.

Selectors are strict by default

To encourage better practices for querying elements, selectors are [strict](#) by default, meaning that it will throw an error if the query returns more than one element.

Don't overload test-utils, inline simple utils

`e2e-test-utils` are too bloated with too many utils. Most of them are simple enough to be inlined directly in tests. With the help of accessible selectors, simple utils are easier to write now. For utils that only take place on a certain page, use Page Object Model instead (with an exception of clearing states with `requestUtils` which are better placed in `e2e-test-utils`). Otherwise, only create an util if the action is complex and repetitive enough.

Favor Page Object Model over utils

As mentioned above, [Page Object Model](#) is the preferred way to create reusable utility functions on a certain page.

The rationale behind using a POM is to group utils under namespaces to be easier to discover and use. In fact, `PageUtils` in the `e2e-test-utils-playwright` package is also a POM, which avoids the need for global variables, and utils can reference each other with `this`.

Restify actions to clear or set states

It's slow to set states manually before or after tests, especially when they're repeated multiple times between tests. It's recommended to set them via API calls. Use `requestUtils.rest` and `requestUtils.batchRest` instead to call the [REST API](#) (and add them to `requestUtils` if needed). We should still add a test for manually setting them, but that should only be tested once.

Avoid global variables

Previously in our Jest + Puppeteer E2E tests, `page` and `browser` are exposed as global variables. This makes it harder to work with when we have multiple pages/tabs in the same test, or if we want to run multiple tests in parallel. `@playwright/test` has the concept of [fixtures](#) which allows us to inject `page`, `browser`, and other parameters into the tests.

Make explicit assertions

We can insert as many assertions in one test as needed. It's better to make explicit assertions whenever possible. For instance, if we want to assert that a button exists before clicking on it, we can do `expect(locator).toBeVisible()` before performing `locator.click()`. This makes the tests flow better and easier to read

Common pitfalls

Overusing snapshots

Cross-browser testing

By default, tests are only run in chromium. You can *tag* tests to run them in different browsers. Use `@browser` anywhere in the test title to run it in that browser. Tests will always run in chromium by default, append `-chromium` to disable testing in chromium. Available browsers are `chromium`, `firefox`, and `webkit`.

```
test( 'I will run in @firefox and @webkit (and chromium by default)', async
      // ...
    } );

test( 'I will only run in @firefox but not -chromium', async ( { page } ) =>
      // ...
    } );

test.describe( 'Grouping tests (@webkit, -chromium)', () => {
  test( 'I will only run in webkit', async ( { page } ) => {
    // ...
  } );
} );
```

First published

March 2, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: End-to-End Testing”](#)

[Previous Testing Overview](#) [Previous: Testing Overview](#)

[Next Migration guide](#) [Next: Migration guide](#)

Migration guide

In this article

Table of Contents

- [Migration steps for tests](#)
- [Migration steps for test utils](#)

[↑ Back to top](#)

This document outlines a typical flow of migrating a Jest + Puppeteer test to Playwright. Note that the migration process is also a good opportunity to refactor or rewrite parts of the tests. Please read the [best practices](#) guide before starting the migration.

Migration steps for tests

1. Choose a test suite to migrate in `packages/e2e-tests/specs`, rename `.test.js` into `.spec.js` and put it in the same folder structure inside `test/e2e/specs`.
2. Require the test helpers from `@wordpress/e2e-test-utils-playwright`:
`js`
`const { test, expect } = require('@wordpress/e2e-test-utils-playwright');`
3. Change all occurrences of `describe`, `beforeAll`, `beforeEach`, `afterEach` and `afterAll` with the `test.` prefix. For instance, `describe` turns into `test.describe`.
4. Use the [fixtures API](#) to require previously global variables like `page` and `browser`.
5. Delete all the imports of `e2e-test-utils`. Instead, use the fixtures API to directly get the `admin`, `editor`, `pageUtils` and `requestUtils`. (However, `admin`, `editor` and `pageUtils` are not allowed in `beforeAll` and `afterAll`, rewrite them using `requestUtils` instead.)
6. If there's a missing util, try to inline the operations directly in the test if there are only a few steps. If you think it deserves to be implemented as a test util, then follow the [guide](#) below.
7. Manually migrate other details in the tests following the proposed [best practices](#). Note that even though the differences in the API of Playwright and Puppeteer are similar, some manual changes are still required.

Migration steps for test utils

Before migrating a test utility function, think twice about whether it's necessary. Playwright offers a lot of readable and powerful APIs which make a lot of the utils obsolete. Try implementing the same thing inline directly in the test first. Only follow the below guide if that doesn't work for you. Some examples of utils that deserve to be implemented in the `e2e-test-utils-playwright` package include complex browser APIs (like `pageUtils.dragFiles` and `pageUtils.pressKeys`) and APIs that set states (`requestUtils.*`).

The `e2e-test-utils-playwright` package is not meant to be a drop-in replacement of the Jest + Puppeteer's `e2e-test-utils` package. Some utils are only created to ease the migration process, but they are not necessarily required.

Playwright utilities are organized a little differently from those in the `e2e-test-utils` package. The `e2e-test-utils-playwright` package has the following folders that utils are divided up into:

- `admin` – Utilities related to WordPress admin or WordPress admin's user interface (e.g. `visitAdminPage`).
- `editor` – Utilities for the block editor (e.g. `clickBlockToolbarButton`).
- `pageUtils` – General utilities for interacting with the browser (e.g. `pressKeys`).
- `requestUtils` – Utilities for making REST API requests (e.g. `activatePlugin`). These utilities are used for setup and teardown of tests.

1. Copy the existing file in `e2e-test-utils` and paste it in the `admin`, `editor`, `page` or `request` folder in `e2e-test-utils-playwright` depending on the type of util.

2. Update any parts of the util that need to be rewritten:
 - The `page` and `browser` variables are available in `admin`, `editor` and `pageUtils` as `this.page` and `this.browser`.
 - All the other utils in the same class are available in `this` and bound to the same instance. You can remove any `import` statements and use `this` to access them.
 - Consider updating the util to use TypeScript if you're comfortable doing so.
3. Import the newly migrated util in `index.ts` and put it inside the `Admin/Editor/PageUtils/RequestUtils` class as an instance field.

First published

March 2, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Migration guide”](#)

[Previous End-to-End Testing](#) [Previous: End-to-End Testing](#)

[Next Overusing snapshots](#) [Next: Overusing snapshots](#)

Overusing snapshots

In this article

[Table of Contents](#)

- [Problems with snapshot testing](#)
- [The solution](#)
- [Snapshot variants](#)
- [What about test coverage?](#)
- [Best practices](#)
 - [Avoid huge snapshots](#)
 - [Avoid repetitive snapshots](#)
- [Further readings](#)

[↑ Back to top](#)

Take a look at the below code. Could you understand what the test is trying to do at first glance?

```
await editor.insertBlock( { name: 'core/quote' } );
await page.keyboard.type( '1' );
await page.keyboard.press( 'Enter' );
await page.keyboard.press( 'Enter' );

expect( await editor.getEditedPostContent() ).toMatchSnapshot();

await page.keyboard.press( 'Backspace' );
```

```
await page.keyboard.type('2');

expect( await editor.getEditedPostContent() ).toMatchSnapshot();
```

This is borrowed from the real code in gutenberg, with the test title and the comments removed and refactored into Playwright. Ideally, E2E tests should be self-documented and readable to end users; in the end, they are trying to resemble how end users interact with the app. However, there are a couple of red flags in the code.

Problems with snapshot testing

Popularized by Jest, [snapshot testing](#) is a great tool to help test our app *when it makes sense*. However, probably because it's so powerful, it's often overused by developers. There are already multiple [articles](#) about this. In this particular case, snapshot testing fails to reflect the developer's intention. It's not clear what the assertions are about without looking into other information. This makes the code harder to understand and creates a mental overhead for all the other readers other than the one who wrote it. As readers, we have to jump around the code to fully understand them. The added complexity of the code discourages contributors from changing the test to fit their needs. It could sometimes even confuse the authors and make them accidentally [commit the wrong snapshots](#).

Here's the same test with the test title and comments. Now you know what these assertions are actually about.

```
it('can be split at the end', async () => {
  // ...

  // Expect empty paragraph outside quote block.
  expect( await getEditedPostContent() ).toMatchSnapshot();

  // ...
  // Expect the paragraph to be merged into the quote block.
  expect( await getEditedPostContent() ).toMatchSnapshot();
});
```

The developer's intention is a bit more readable, but it still feels disconnected from the test. You might be tempted to try [inline snapshots](#), which do solve the issue of having to jump around files, but they're still not self-documented nor explicit. We can do better.

The solution

Instead of writing the assertions in comments, we can try directly writing them out explicitly. With the help of `editor.getBlock`s, we can rewrite them into simpler and atomic assertions.

```
// ...

// Expect empty paragraph outside quote block.
await expect.poll( editor.getBlock() ).toMatchSnapshot(
  {
    name: 'core/quote',
    innerBlocks: [
      {
```

```

        name: 'core/paragraph',
        attributes: { content: '1' },
    },
],
{
    name: 'core/paragraph',
    attributes: { content: '' },
}
] );
// ...

// Expect the paragraph to be merged into the quote block.
await expect.poll( editor.getBlocks ).toMatchObject( [ {
    name: 'core/quote',
    innerBlocks: [
        {
            name: 'core/paragraph',
            attributes: { content: '1' },
        },
        {
            name: 'core/paragraph',
            attributes: { content: '2' },
        },
    ],
} ] );

```

These assertions are more readable and explicit. You can add additional assertions or split existing ones into multiple ones to highlight their importance. Whether to keep the comments is up to you, but it's usually fine to omit them when the code is already readable without them.

Snapshot variants

Due to the lack of inline snapshots in Playwright, some migrated tests are using string assertions (`toBe`) to simulate similar effects without having to create dozens of snapshot files.

```
expect( await editor.getEditedPostContent() ).toBe( `<!-- wp:paragraph -->
<p>Paragraph</p>
<!-- /wp:paragraph -->` );
```

We can consider this pattern as a variant of snapshot testing, and we should follow the same rule when writing them. It's often better to rewrite them using `editor.getBlocks` or other methods to make explicit assertions.

```
await expect.poll( editor.getBlocks ).toMatchObject( [ {
    name: 'core/paragraph',
    attributes: { content: 'Paragraph' },
} ] );
```

What about test coverage?

Comparing the explicit assertions to snapshot testing, we're definitely losing some test coverage in this test. Snapshot testing is still useful when we want to assert the full serialized content of the block. Fortunately, though, some tests in the integration test already assert the [full content](#) of each core block. They run in Node.js, making them way faster than repeating the same test in Playwright. Running 273 test cases in my machine only costs about 5.7 seconds. These sorts of tests work great at the unit or integration level, and we can run them much faster without losing test coverage.

Best practices

Snapshot testing should rarely be required in E2E tests, often there are better alternatives that leverage explicit assertions. For times when there isn't any other suitable alternative, we should follow the best practices when using them.

Avoid huge snapshots

Huge snapshots are hard to read and difficult to review. Moreover, when everything is important then nothing is important. Huge snapshots prevent us from focusing on the important parts of the snapshots.

Avoid repetitive snapshots

If you find yourself creating multiple snapshots of similar contents in the same test, then it's probably a sign that you want to make more atomic assertions instead. Rethink what you want to test, if the first snapshot is only just a reference for the second one, then what you want is likely the **difference** between the snapshots. Store the first result in a variable and assert the difference between the results instead.

Further readings

- [Effective Snapshot Testing – Kent C. Dodds](#)
- [Common Testing Mistakes – Kent C. Dodds](#)

First published

March 2, 2023

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Overusing snapshots”](#)

Scripts

In this article

Table of Contents

- [WordPress scripts](#)
- [Vendor scripts](#)
- [Polyfill scripts](#)
- [Bundling and code sharing](#)

[↑ Back to top](#)

The editor provides several vendor and internal scripts to plugin developers. Script names, handles, and descriptions are documented in the table below.

[WordPress scripts](#)

The editor includes a number of packages to enable various pieces of functionality. Plugin developers can utilize them to create blocks, editor plugins, or generic plugins.

Script Name	Handle	Description
Blob	wp-blob	Blob utilities
Block Library	wp-block-library	Block library for the editor
Blocks	wp-blocks	Block creations
Block Serialization	wp-block-serialization	Default block serialization parser
Default Parser	default-parser	implementations for WordPress documents
Block Serialization Spec	wp-block-serialization-spec	Grammar file (grammar.pegjs) for WordPress posts
Parser	parser	
Components	wp-components	Generic components to be used for creating common UI elements
Compose	wp-compose	Collection of handy Higher Order Components (HOCs)
Core Data	wp-core-data	Simplify access to and manipulation of core WordPress entities
Data	wp-data	Data module serves as a hub to manage application state for both plugins and WordPress itself
Date	wp-date	Date module for WordPress
Deprecated	wp-deprecated	Utility to log a message to notify developers about a deprecated feature
Dom	wp-dom	DOM utilities module for WordPress
Dom Ready	wp-dom-ready	Execute callback after the DOM is loaded
Editor	wp-editor	Building blocks for WordPress editors
Edit Post	wp-edit-post	Edit Post Module for WordPress

Script Name	Handle	Description
Element	wp-element	Element is, quite simply, an abstraction layer atop React
Escape Html	wp-escape-html	Escape HTML utils
Hooks	wp-hooks	A lightweight and efficient EventManager for JavaScript
Html Entities	wp-html-entities	HTML entity utilities for WordPress
I18N	wp-i18n	Internationalization utilities for client-side localization
Is Shallow Equal	wp-is-shallow-equal	A function for performing a shallow comparison between two objects or arrays
Keycodes	wp-keycodes	Keycodes utilities for WordPress, used to check the key pressed in events like <code>onKeyDown</code>
List Reusable blocks	wp-list-reusable-blocks	Package used to add import/export links to the listing page of the reusable blocks
NUX	wp-nux	Components, and <code>wp.data</code> methods useful for onboarding a new user to the WordPress admin interface
Plugins	wp-plugins	Plugins module for WordPress
Redux Routine	wp-redux-routine	Redux middleware for generator coroutines
Rich Text	wp-rich-text	Helper functions to convert HTML or a DOM tree into a rich text value and back
Shortcode	wp-shortcode	Shortcode module for WordPress
Token List	wp-token-list	Constructable, plain JavaScript DOMTokenList implementation, supporting non-browser runtimes
URL	wp-url	A collection of utilities to manipulate URLs
Viewport	wp-viewport	Module for responding to changes in the browser viewport size
Wordcount	wp-wordcount	WordPress word count utility

[**Vendor scripts**](#)

The editor also uses some popular third-party packages and scripts. Plugin developers can use these scripts as well without bundling them in their code (and increasing file sizes).

Script Name	Handle	Description
React	react	React is a JavaScript library for building user interfaces
React Dom	react-dom	Serves as the entry point to the DOM and server renderers for React, intended to be paired with React
Moment	moment	Parse, validate, manipulate, and display dates and times in JavaScript
Lodash	lodash	Lodash is a JavaScript library which provides utility functions for common programming tasks

Polyfill scripts

The editor also provides polyfills for certain features that may not be available in all modern browsers.

It is recommended to use the main `wp-polyfill` script handle which takes care of loading all the below mentioned polyfills.

Script Name	Handle	Description
Babel Polyfill	<code>wp-polyfill</code>	Emulate a full ES2015+ environment. Main script to load all the below mentioned additional polyfills
Fetch Polyfill	<code>wp-polyfill-fetch</code>	Polyfill that implements a subset of the standard Fetch specification
Promise Polyfill	<code>wp-polyfill-promise</code>	Lightweight ES6 Promise polyfill for the browser and node
Formdata Polyfill	<code>wp-polyfill-formdata</code>	Polyfill conditionally replaces the native implementation
Node Contains Polyfill	<code>wp-polyfill-node-contains</code>	Polyfill for Node.contains
Element Closest Polyfill	<code>wp-polyfill-element-closest</code>	Return the closest element matching a selector up the DOM tree

Bundling and code sharing

When using a JavaScript bundler like [webpack](#), the scripts mentioned here can be excluded from the bundle and provided by WordPress in the form of script dependencies see [wp_enqueue_script](#).

The [@wordpress/dependency-extraction-webpack-plugin](#) provides a webpack plugin to help extract WordPress dependencies from bundles. The [@wordpress/scripts build](#) script includes the plugin by default.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Scripts”](#)

[Previous Overusing snapshots](#) [Previous: Overusing snapshots](#)
[Next Managing Packages](#) [Next: Managing Packages](#)

Managing Packages

[↑ Back to top](#)

This repository uses [monorepo](#) to manage WordPress modules and publish them with [lerna](#) as packages to [npm](#). This enforces certain steps in the workflow which are described in details in [packages](#) documentation.

Maintaining dozens of npm packages is difficult—it can be tough to keep track of changes. That's why we use `CHANGELOG.md` files for each package to simplify the release process. As a contributor, you should add an entry to the aforementioned file each time you contribute adding production code as described in [Maintaining Changelogs](#) section.

Publishing WordPress packages to npm is automated by synchronizing it with the bi-weekly Gutenberg plugin RC1 release. You can learn more about this process and other ways to publish new versions of npm packages in the [Gutenberg Release Process document](#).

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Managing Packages”](#)

[Previous Scripts](#) [Previous: Scripts](#)

[Next Gutenberg Release Process](#) [Next: Gutenberg Release Process](#)

Gutenberg Release Process

In this article

Table of Contents

- [Gutenberg plugin releases](#)
 - [Release schedule](#)
 - [Release management](#)
 - [Preparing a release](#)
 - [Documenting the release](#)
 - [Creating minor releases](#)
- [Packages releases to NPM and WordPress Core updates](#)
 - [Synchronizing the Gutenberg plugin](#)
 - [WordPress releases](#)
 - [Standalone bugfix package releases](#)
 - [Development releases](#)

[↑ Back to top](#)

The [Gutenberg repository](#) on GitHub is used to perform several types of releases. This document serves as a checklist for each of these releases, and it can help you understand the different workflows involved.

Before you begin, there are some requirements that must be met in order to successfully release a stable version of the Gutenberg plugin. You will need to:

- Be a member of the [Gutenberg development team](#). This gives you the ability to launch the GitHub actions that are related to the release process and to backport pull requests (PRs) to the release branch.
- Have write permissions on the [Make WordPress Core](#) blog. This allows you to draft the release post.
- Obtain approval from a member of the Gutenberg Core team in order to upload the new version Gutenberg to the WordPress.org plugin directory.

Similar requirements apply to releasing WordPress's [npm packages](#).

[Gutenberg plugin releases](#)

The first step in releasing a stable version of the Gutenberg plugin is to [create an issue](#) in the Gutenberg repository. The issue template is called “Gutenberg Release,” and it contains a checklist for the complete release process, from release candidate to changelog curation to cherry-picking, stable release, and release post. The issue for [Gutenberg 15.7](#) is a good example.

The checklist helps you coordinate with developers and other teams involved in the release process. It ensures that all of the necessary steps are completed and that everyone is aware of the schedule and important milestones.

[Release schedule](#)

A new major version of Gutenberg is released approximately every two weeks. The current and next versions are tracked in [GitHub milestones](#), along with the date when each version will be tagged.

On the date of the current milestone, also called the tagging date, the first release candidate (RC) of Gutenberg is published. This is a pre-release version of the plugin that is made available for testing by plugin authors and users. If any regressions are found, a new RC can be published.

Release candidates are versioned incrementally, starting with `-rc.1`, then `-rc.2`, and so on. As soon as the first RC (RC1) is published, preparation for the release post begins.

One week after the RC1, the stable version is created based on the last RC and any necessary regression fixes. Once the stable version is released, the release post is published.

If critical bugs are discovered in stable versions of the plugin, patch versions can be released at any time.

Release management

Each major Gutenberg release is run by a release manager, also known as a release lead. This individual, or small team of individuals, is responsible for the release of Gutenberg with support from the broader [Gutenberg development team](#).

The release manager is responsible for initiating all release activities, and their approval is required for any changes to the release plan. In the event of an emergency or if the release manager is unavailable, other team members may take appropriate action, but they should keep the release manager informed.

If you are a member of the [Gutenberg development team](#) and are interested in leading a Gutenberg release, reach out in the [#core-editor](#) Slack channel.

Preparing a release

The plugin release process is mostly automated and happens on GitHub. You do not need to run any steps locally on your machine. However, it's a good idea to have a local copy of Gutenberg for changelog preparation, general testing, and in case multiple release candidates are required. But more on that later.

Here is an [11-minute video](#) that demonstrates the plugin release process. If you are unfamiliar with the process, we recommend watching the video first. The process is also documented in the following paragraphs, which provide more detailed instructions.

Organizing and labeling milestone PRs

Quick reference

- Ensure all PRs are properly labeled.
- Each PR must have one label prefixed by [Type].

The first step in preparing a Gutenberg release is to organize all PRs assigned to the current [milestone](#) and ensure that each is properly labeled. [Labels](#) are used to automatically generate the changelog, and changing the labels on PRs is much faster than reorganizing an existing changelog in the release section afterward.

To test the changelog automation that will be run as part of the release workflow, you can use the following command in your local copy of Gutenberg using the milestone of the stable release version you are working on:

```
npm run other:changelog -- --milestone="Gutenberg 16.2"
```

The output of this command is the changelog for the provided milestone, which in the above example is Gutenberg 16.2. You can copy and paste the output into a Markdown document, which will make it easier to view and allow you to follow the links to each PR.

All PRs should have a label prefixed by [Type] as well as labels for sub-categories. The two most common labels are [Type] Bug and [Type] Enhancement. When reviewing the generated changelog, pay close attention to the following:

- **Enhancements:** Look for PRs that don't have any subcategories attached.
- **Bug fixes:** Also look for PRs that don't have any subcategories attached.
- **Various:** PRs in this section don't have any labels at all.

Update the labels on each PR as needed. You can continue generating the changelog until you are comfortable proceeding. Now you are ready to start the release candidate workflow.

You can see how the changelog is generated from the PR labels in the [changelog.js](#) file.

Running the release workflow

Quick reference

- Announce in [#core-editor](#) that you are about to start the release workflow.
- Run the [Build Gutenberg Plugin Zip](#) workflow.

Before you begin, announce in [#core-editor](#) Slack channel that you are about to start the workflow and indicate whether you are releasing a stable version of Gutenberg or an RC.

Then go to the Gutenberg repository, click on the Actions tab, and then locate the [Build Gutenberg Plugin Zip](#) action. Note the blue banner that says, “This workflow has a `workflow_dispatch` event trigger.” Expand the “Run workflow” dropdown on its right-hand side.

The screenshot shows the GitHub Actions workflow runs page for the 'Build Gutenberg Plugin Zip' workflow. At the top, it displays '153 workflow runs'. Below this, a blue banner states: 'This workflow has a `workflow_dispatch` event trigger.' Two workflow runs are listed:

- Fix download-artifact action path arg**
Build Gutenberg Plugin Zip #153: Commit b6dfd4f
pushed by ockham
- Try without escape characters**
Build Gutenberg Plugin Zip #152: Commit 55831ec
pushed by ockham

To the right of the runs, there is a configuration sidebar with the following options:

- Use workflow from**: A dropdown set to 'Branch: trunk'.
- rc or stable? ***: A text input field.
- Run workflow**: A green button.

To release an RC version of the plugin, enter `rc` in the text field. To release a stable version, enter `stable`. In each case, press the button “Run workflow”.

This will trigger a GitHub Actions (GHA) workflow that will bump the plugin version, build the Gutenberg plugin `.zip` file, create a release draft, and attach the plugin `.zip` file. This part of the process typically takes about six minutes. The workflow will appear at the top of the list, right under the blue banner. Once it is finished, the workflow’s status icon will change from a yellow dot to a green checkmark. You can follow along for a more detailed view by clicking on the workflow.

Publishing the @wordpress packages to NPM

As part of the release workflow, all of the @wordpress packages are published to NPM. After the [Build Gutenberg Plugin Zip](#) action has created the draft release, you may see a message that the [Publish npm packages](#) action requires someone with appropriate permissions to trigger it.

This message is misleading. You do not need to take any action to publish the @wordpress packages to NPM. The process is automated and will automatically run after the release notes are published.

Viewing the release draft

As soon as the workflow has finished, you'll find the release draft under [Gutenberg Releases](#). The draft is pre-populated with changelog entries based on previous RCs for this version and any changes that have since been cherry-picked to the release branch. Thus, when releasing the first stable version of a series, delete any RC version headers (that are only there for your information) and move the more recent changes to the correct section (see below).

Curating the release changelog

The best time to work on the changelog is when it is first created during the release candidate workflow. This is when the changelog automation is called, and the first version of the changelog becomes available. The changelog process is mostly automated, but it depends heavily on the proper labeling of the PRs in the milestone, as mentioned above.

The stable release process takes the changelogs of the RCs and adds them to the stable release. However, there is one important thing to note: the stable release only “remembers” the first version of the changelog, which is the version that was available when RC1 was published. Any subsequent changes to the changelog of RC1 will not be included in the stable release.

That means if you curate the whole changelog before you publish RC1, you won't have to work on it for the stable release, except for the few items of subsequent RC2 or RC3 releases that will also be added to the stable release.

Once the release changelog is available in the draft, take some time to read the notes and edit them to make sure they are easy to read and accurate. Don't rush this part. It's important to make sure the release notes are as organized as possible, but you don't have to finish them all at once. You can save the draft and come back to it later.

If you're worried that people won't be able to access the release candidate version until you publish the release, you can share the release artifact with the [#core-editor](#) Slack channel. This will give people access to the release candidate version while you finish curating the changelog.

Here are some additional tips for preparing clear and concise changelogs:

- Move all entries under the `Various` section to a more appropriate section.
- Fix spelling errors or clarify wording. Phrasing should be easy to understand where the intended audience is those who use the plugin or are keeping up with ongoing development.
- Create new groupings as applicable, and move pull requests between.
- When multiple PRs relate to the same task (such as a follow-up pull request), try to combine them into a single entry. Good examples for this are PRs around removing Lodash for performance purposes, replacement of Puppeteer E2D tests with Playwright or efforts to convert public components to TypeScript.

- If subtasks of a related set of PRs are substantial, consider organizing as entries in a nested list.
- Remove PRs that revert other PRs in the same release if the net change in code is zero.
- Remove all PRs that only update the mobile app. The only exception to this rule is if the mobile app pull request also updates functionality for the web.
- If a subheader only has one PR listed, remove the subheader and move the PR to the next matching subheader with more than one item listed.

Creating release candidate patches (cherry-picking)

Quick reference

- Ensure all PRs that need cherry-picking have the `Backport to Gutenberg` RC label.
- In your local clone of the Gutenberg repository, switch to the release branch: `git checkout release/X.Y`
- Cherry-pick all merged PRs using the automated script: `npm run other:cherry-pick "Backport to Gutenberg RC"`

After an RC is published but before the final stable release, some bugs related to the release might be fixed and committed to `trunk`. The stable release will not automatically include these fixes. Including them is a manual process, which is called cherry-picking.

There are a couple of ways you might be made aware of these bugs as a release manager:

- Contributors may add the `Backport to Gutenberg` RC label to a closed PR. [Do a search for any of these PRs](#) before publishing the final release.
- You may receive a direct message or a ping in the [#core-editor](#) Slack channel notifying you of PRs that need to be included in the RC. Even when this is the case, the `Backport to Gutenberg` RC should always be added to the PR.

Automated cherry-picking

The cherry-picking process can be automated with the `npm run other:cherry-pick "[Insert Label]"` script, which is included in Gutenberg. You will need to use the label `Backport to Gutenberg` RC when running the command and ensure all PRs that need cherry-picking have the label assigned.

To cherry-pick PRs, you must clone (not fork) the Gutenberg repository and have write access. Only members of the [Gutenberg development team](#) have the necessary permissions to perform this action.

Once you have cloned the Gutenberg repository to your local development environment, begin by switching to the release branch:

```
git checkout release/X.Y
```

Next, cherry-pick all the merged PRs with the appropriate backport label:

```
npm run other:cherry-pick "Backport to Gutenberg RC"
```

Behind the scenes, the script will:

- Cherry-pick all PRs with the label `Backport to Gutenberg` RC
- Add them to the release milestone

- `git push` all changes to the release branch
- Add a comment to the PR indicating it's been cherry-picked
- Remove the label `Backport to Gutenberg RC` from the PR

Here is a screenshot of the process:



PROBLEMS

DEBUG CONSOLE

TERMINAL



TERMINAL

```
Switched to branch 'release/15.7'  
Your branch is up to date with 'origin/
```

```
o pauli@Birgits-MBP gutenberg % npm
```

```
> gutenberg@15.7.0-rc.1 other:cher  
> node ./bin/cherry-pick.mjs "Back
```

```
You are on branch "release/15.7".  
This script will:
```

- Cherry-pick the merged PRs labeled
- Ask whether you want to push this
- Comment on each PR
- Remove the label from each PR

```
The last two actions will be performed  
you've linked to your GitHub CLI (
```

```
Do you want to proceed? (Y/n)Y
```

```
$ git pull origin release/15.7 --rebase
```

```
$ git fetch origin trunk...
```

```
Found the following PRs to cherry-
```

```
#50242 – Fix site logo preview
```

```
Fetching commit IDs...
```

```
Done!
```

```
#50242 – a284c976079801450e8d –
```

```
Trying to cherry-pick one by one...
```

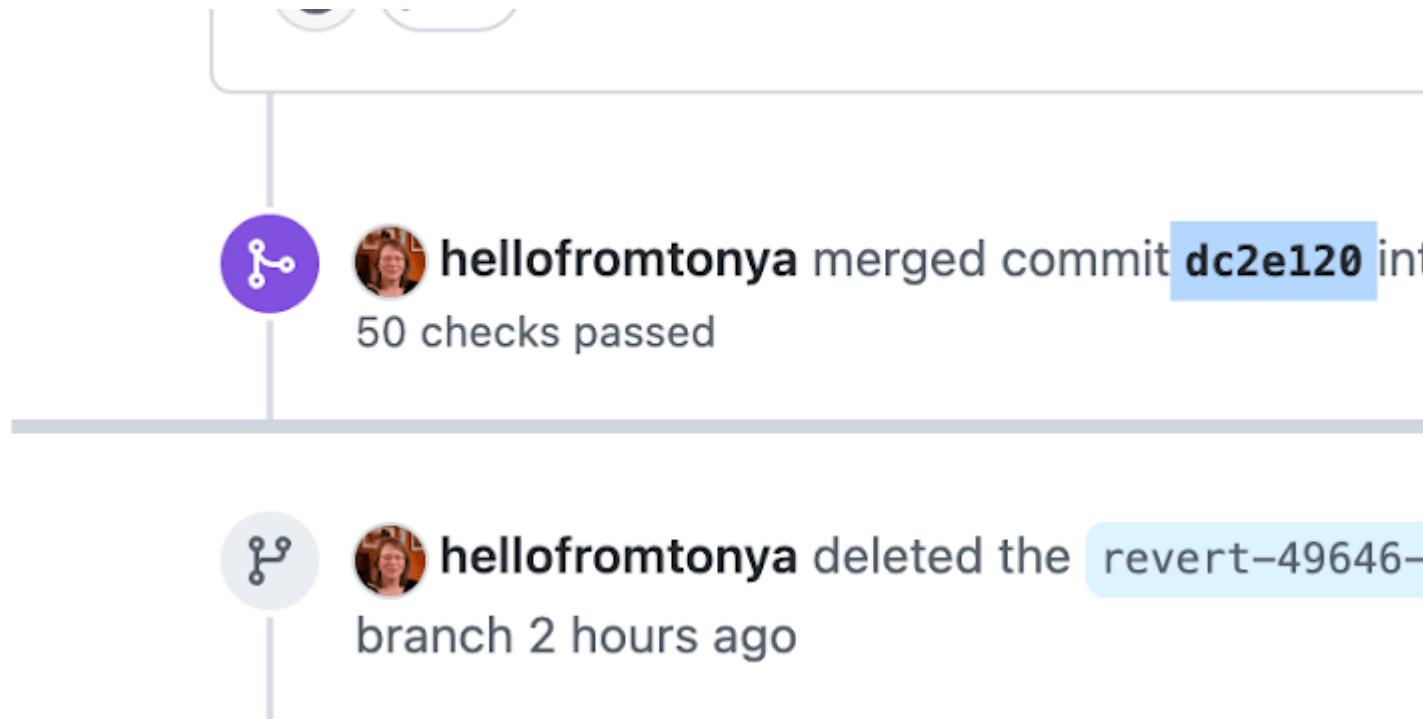
```
Cherry-picking round 1:
```

Manual cherry-picking

If you need to handle cherry-picking one at a time and one step at a time, you can follow this sequence manually. After checking out the corresponding release branch:

1. Cherry-pick each PR (in chronological order) using `git cherry-pick [SHA]`.
2. When done, push the changes to GitHub using `git push`.
3. Remove the `Backport to Gutenberg RC` label and update the milestone to the current release for all cherry-picked PRs.

To find the [SHA] for a pull request, open the PR, and you'll see a message “[Username] merged commit [SHA] into trunk” near the end.



If the cherry-picked fixes deserve another release candidate before the stable version is published, create one by following the instructions above. Let other contributors know that a new release candidate has been released in the [#core-editor](#) Slack channel.

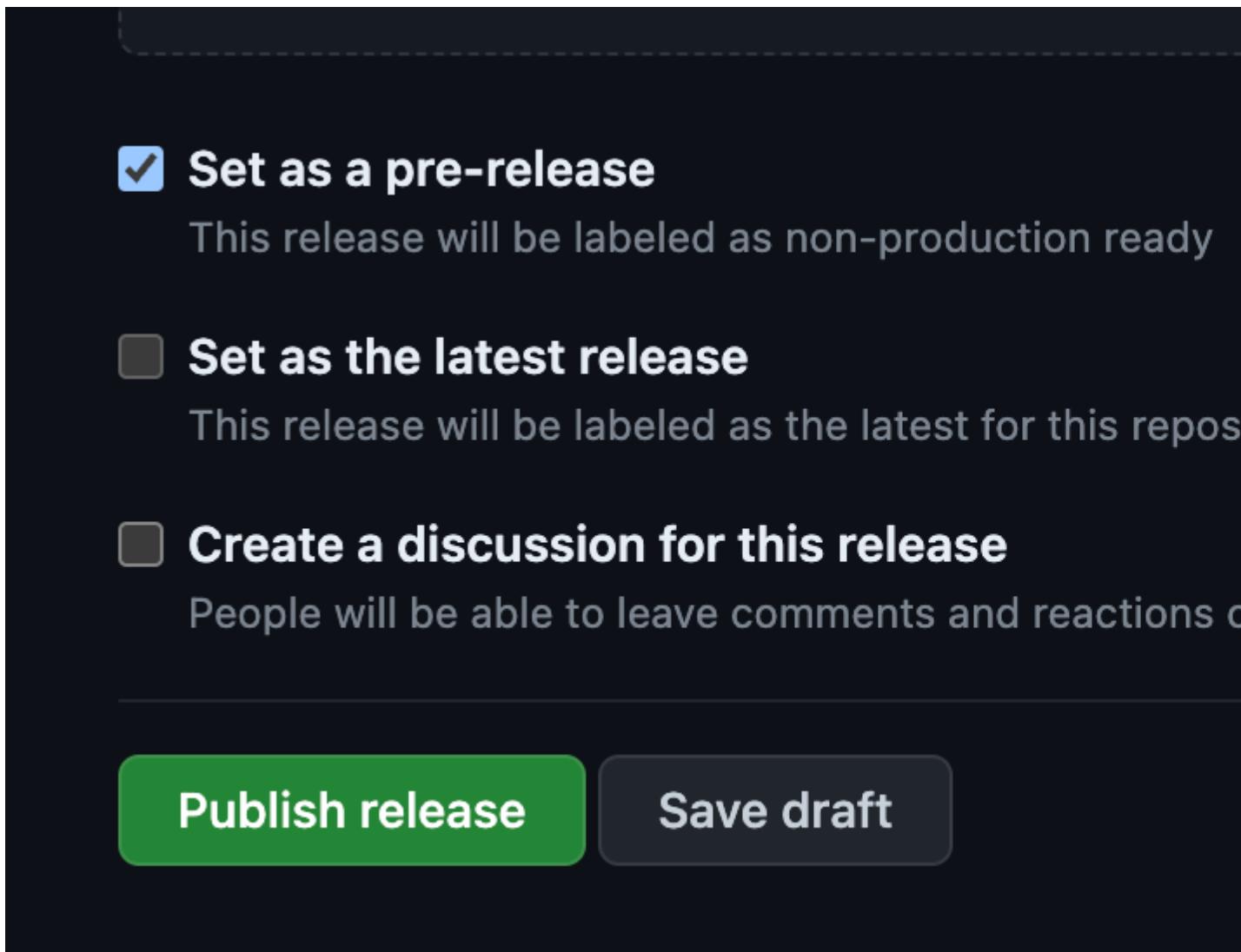
Publishing the release

Quick reference

- In the release draft, press the “Publish release” button.
- If publishing a stable release, get approval from a member of the [Gutenberg Release](#), [Gutenberg Core](#), or the [WordPress Core](#) teams to upload the new plugin version to the WordPress.org plugin repository (SVN).
- Once uploaded, confirm that the latest version can be downloaded and updated from the WordPress plugin dashboard.

Only once you’re happy with the shape of the changelog in the release draft, press the “Publish release” button.

Note that you do not need to change the checkboxes above the button. If you are publishing an RC, the “Set as a pre-release” will automatically be selected, and “Set as the latest release” will be selected if you are publishing the stable version.



Publishing the release will create a `git` tag for the version, publish the release, and trigger [another GHA workflow](#) with a twofold purpose:

1. Use the release notes that you just edited to update `changelog.txt`, and
2. Upload the new plugin version to the WordPress.org plugin repository (SVN) (only if you’re releasing a stable version).

The last step needs approval by a member of either the [Gutenberg Release](#), [Gutenberg Core](#), or the [WordPress Core](#) teams. These teams get a notification email when the release is ready to be approved, but if time is of the essence, you can ask in the `#core-editor` Slack channel or ping the [Gutenberg Release team](#)) to accelerate the process. Reaching out before launching the release process so that somebody is ready to approve is recommended. Locate the “[Upload Gutenberg plugin to WordPress.org plugin repo](#)” workflow for the new version, and have it [approved](#).

Once approved, the new Gutenberg version will be available to WordPress users all over the globe. Once uploaded, confirm that the latest version can be downloaded and updated from the WordPress plugin dashboard.

The final step is to write a release post on make.wordpress.org/core. You can find some tips on that below.

Troubleshooting the release

The plugin was published to the WordPress.org plugin directory but the workflow failed.

This has happened occasionally, see [this one](#) for example.

It's important to check that:

- the plugin from the directory works as expected
- the ZIP contents (see [Downloads](#)) looks correct (doesn't have anything obvious missing)
- the [Gutenberg SVN repo](#) has two new commits (see [the log](#)):
 - the `trunk` folder should have "Committing version X.Y.Z"
 - there is a new `tags/X.Y.Z` folder with the same contents as `trunk` whose latest commit is "Tagging version X.Y.Z"

Most likely, the tag folder couldn't be created. This is a [known issue](#) that [can be fixed manually](#).

Either substitute `SVN_USERNAME`, `SVN_PASSWORD`, and `VERSION` for the proper values or set them as global environment variables first:

```
# CHECKOUT THE REPOSITORY
svn checkout https://plugins.svn.wordpress.org/gutenberg/trunk --username
# MOVE TO THE LOCAL FOLDER
cd gutenberg-svn

# IF YOU HAPPEN TO HAVE ALREADY THE REPO LOCALLY
# AND DIDN'T CHECKOUT, MAKE SURE IT IS UPDATED
# svn up .

# COPY CURRENT TRUNK INTO THE NEW TAGS FOLDER
svn copy https://plugins.svn.wordpress.org/gutenberg/trunk https://plugins
```

Ask around if you need help with any of this.

Documenting the release

Documenting the release is led by the release manager with the help of [Gutenberg development team](#) members. This process is comprised of a series of sequential steps that, because of the number of people involved, and the coordination required, need to adhere to a timeline between the RC and stable releases. Stable Gutenberg releases happen on Wednesdays, one week after the initial RC.

Timeline

1. Make a copy of the [Google Doc Template for release posts](#) – Wednesday to Friday
2. Select the release highlights – Friday to Monday
3. Request release assets (images, videos) from the Design team once highlights are finalized – Friday to Monday
4. Draft the release post and request peer review – Monday to Wednesday

5. Publishing the post after the stable version is released – Wednesday

Selecting the release highlights

Once the changelog is cleaned up, the next step is to choose a few changes to highlight in the release post. These highlights usually focus on new features and enhancements, including performance and accessibility ones, but can also include important API changes or critical bug fixes.

Given the big scope of Gutenberg and the high number of PRs merged in each milestone, it is not uncommon to overlook impactful changes worth highlighting; because of this, this step is a collaborative effort between the release manager and other Gutenberg development team members. If you don't know what to pick, reach out to others on the team for assistance.

Requesting release assets

After identifying the highlights of a new WordPress release, the release manager requests visual assets from the Design team. The request is made in the [#design](#) Slack channel, and an example post for 15.8 can be found [here](#). The assets will be provided in a [Google Drive folder](#) assigned to the specific release.

When creating visual assets for a WordPress release, use animations (video or GIF) or static images to showcase the highlights. Use [previous release posts](#) as a guide, and keep in mind that animations are better for demonstrating workflows, while more direct highlights can be shown with an image. When creating assets, avoid using copyrighted material and disable browser plugins that can be seen in the browser canvas.

Drafting the release post

The release manager is responsible for drafting the release post based on the [Google Doc Template](#). That said, because of the nature of the release post content, responsibilities can be divided up and delegated to other team members if agreed upon in advance. Once the draft is complete, ask for peer review.

Publishing the release post

Once the post content is ready, an author with permission to post on [make.wordpress.org/core](#) will create a new draft and import the content. The post should include the following tags:

- [#block-editor](#)
- [#core-editor](#)
- [#gutenberg](#)
- [#gutenberg-new](#)

The author should then enable public preview on the post and ask for a final peer review. This is encouraged by the [make/core posting guidelines](#).

Finally, the post should be published after the stable version is released and is available on WordPress.org. This will help external media to echo and amplify the release.

Creating minor releases

Occasionally it's necessary to create a minor release (i.e. X.Y.Z) of the Plugin. This is usually done to expedite fixes for bad regressions or bugs. The [Backport to Gutenberg Minor Release](#) is usually used to identify PRs that need to be included in a minor release, but as release coordinator you may also be notified more informally through slack. Even so, it's good to ensure all relevant PRs have the correct label.

As you proceed with the following process, it's worth bearing in mind that such minor releases are not created as branches in their own right (e.g. `release/12.5.0`) but are simply [tags](#).

The method for minor releases is nearly identical to the main Plugin release process (see above) but has some notable exceptions. Please make sure to read *the entire* guide before proceeding.

Updating the release branch

The minor release should only contain the *specific commits* required. To do this you should checkout the previous *major* stable (i.e. non-RC) release branch (e.g. `release/12.5`) locally and then cherry pick any commits that you require into that branch.

If an RC already exists for a new version, you *need* to cherry-pick the same commits in the respective release branch, as they will not be included automatically. E.g.: If you're about to release a new minor release for 12.5 and just cherry-picked into `'release/12.5'`, but 12.6.0-rc.1 is already out, then you need to cherry-pick the same commits into the `'release/12.6'` branch, or they won't be included in subsequent releases for 12.6! Usually it's best to coordinate this process with the release coordinator for the next release.

The cherry-picking process can be automated with the [`npm run cherry-pick`](#) script, but be sure to use the [Backport to Gutenberg Minor Release](#) label when running the script.

You must also ensure that all PRs being included are assigned to the Github Milestone on which the minor release is based. Bear in mind, that when PRs are *merged* they are automatically assigned a milestone for the next *stable* release. Therefore you will need to go back through each PR in Github and re-assign the Milestone.

For example, if you are releasing version 12.5.4, then all PRs picked for that release must be unassigned from the 12.6 Milestone and instead assigned to the 12.5 Milestone.

Once cherry picking is complete, you can also remove the [Backport to Gutenberg Minor Release](#) label from the PRs.

Once you have the stable release branch in order and the correct Milestone assigned to your PRs you can *push the branch to Github* and continue with the release process using the Github website GUI.

Running the minor release

This workflow has a `workflow_dispatch` event trigger.

Fix download-artifact action path arg
Build Gutenberg Plugin Zip #153: Commit b6dfd4f pushed by ockham

Try without escape characters
Build Gutenberg Plugin Zip #152: Commit 55831ec pushed by ockham

Use workflow from
Branch: trunk

rc or stable? *

Run workflow

Go to Gutenberg’s GitHub repository’s Actions tab, and locate the [“Build Gutenberg Plugin Zip” action](#). You should now *carefully* choose the next action based on information about the current Plugin release version:

If the previous release version was **stable** (X.Y.Z – e.g. 12.5.0, 12.5.1 .etc) leave the Use workflow from field as `trunk` and then specify `stable` in the text input field. The workflow will automatically create a minor release, with z incremented (x.y.(z+1)) as required.

If however, the previous release was an **RC** (e.g. X.Y.0 - `rc` .1) you will need to *manually* select the *stable version’s release branch* (e.g. 12.5.0) when creating the release. Failure to do this will cause the workflow to release the next major *stable* version (e.g. 12.6.0) which is not what you want.

To do this, when running the Workflow, select the appropriate `release/` branch from the Use workflow from dropdown (e.g. `release/12.5`) and specify `stable` in the text input field.

Creating a minor release for previous stable releases

It is possible to create a minor release for any release branch even after a more recent stable release has been published. This can be done for *any* previous release branches, allowing more flexibility in delivering updates to users. In the past, users had to wait for the next stable release, potentially taking days. Now, fixes can be swiftly shipped to any previous release branches as required.

The process is identical to the one documented above when an RC is already out: choose a previous release branch, type `stable`, and click “Run workflow”. The release will be published

on the GitHub releases page for Gutenberg and to the WordPress core repository SVN as a tag under <https://plugins.svn.wordpress.org/gutenberg/tags/>. The SVN trunk directory will not be touched.

IMPORTANT: When publishing the draft created by the “[Build Plugin Zip](#)” workflow, make sure to leave the “Set as last release” checkbox unchecked. If it is left checked by accident, the “[Upload Gutenberg plugin to WordPress.org plugin](#)” workflow will still correctly upload it **as a tag (and will not replace the trunk version)** to the WordPress plugin repository SVN – the workflow will perform some version arithmetic to determine how the plugin should be shipped – but you’ll still need to fix the state on GitHub by setting the right release as `latest` on the [releases](#) page!

Troubleshooting

The release draft was created but it was empty/contained an error message

If you forget to assign the correct Milestone to your cherry picked PR(s) then the changelog may not be generated as you would expect.

It is important to always manually verify that the PRs shown in the changelog match up with those cherry picked to the release branch.

Moreover, if the release includes only a single PR, then failing to assign the PR to the correct Milestone will cause an error to be displayed when generating the changelog. In this case you can edit the release notes to include details of the missing PR (manually copying the format from a previous release).

If for any reason the Milestone has been closed, you may reopen it for the purposes of the release.

The draft release only contains 1 asset file. Other releases have x3.

This is expected. The draft release will contain only the plugin zip. Only once the release is published will the remaining assets be generated and added to the release.

Do I need to publish point releases to WordPress.org?

Yes. The method for this is identical to the main Plugin release process. You will need a member of the Gutenberg Core team the Gutenberg Release team to approve the release workflow.

The release process failed to cherry-pick version bump commit to the trunk branch.

First, confirm that the step failed by checking the latest commits on `trunk` do not include the version bump commit. Then revert the version bump commit on the release branch – `git revert --no-edit {commitHash}`. Finally, push the changes and start the release process again.

Packages releases to NPM and WordPress Core updates

The Gutenberg repository follows the [WordPress SVN repository’s](#) branching strategy for every major WordPress release. In addition to that, it also contains two other special branches that control npm publishing workflows:

- The `wp/latest` branch contains the same version of packages as those published to npm with the `latest` distribution tag. The goal here is to have this branch synchronized with

the last Gutenberg plugin release, and the only exception would be an unplanned [bugfix release](#).

- The `wp/next` branch contains the same version of packages as those published to npm with the `next` distribution tag. It always gets synchronized with the `trunk` branch. Projects should use those packages for development or testing purposes only.
- A Gutenberg branch `wp/X.Y` (example `wp/6.2`) targeting a specific WordPress major release (including its further minor increments) gets created based on the current Gutenberg plugin release branch `release/X.Y` (example `release/15.1`) shortly after the last release planned for inclusion in the next major WordPress release.

Release types and their schedule:

- [Synchronizing Gutenberg Plugin](#) (`latest` dist tag) – publishing happens automatically every two weeks based on the newly created `release/X.Y` (example `release/12.8`) branch with the RC1 version of the Gutenberg plugin.
- [WordPress Releases](#) (`wp-X.Y` dist tag, example `wp-6.2`) – publishing gets triggered on demand from the `wp/X.Y` (example `wp/6.2`) branch. Once we reach the point in the WordPress major release cycle (shortly before Beta 1) where we only cherry-pick commits from the Gutenberg repository to the WordPress core, we use `wp/X.Y` branch (created from `release/X.Y` branch, example `release/15.1`) for npm publishing with the `wp-X.Y` dist-tag. It's also possible to use older branches to backport bug or security fixes to the corresponding older versions of WordPress Core.
- [Development Releases](#) (`next` dist tag) – it is also possible to perform development releases at any time when there is a need to test the upcoming changes.

There is also an option to perform [Standalone Bugfix Package Releases](#) at will. It should be reserved only for critical bug fixes or security releases that must be published to `npm` outside of regular cycles.

[Synchronizing the Gutenberg plugin](#)

For each Gutenberg plugin release, we also publish to npm an updated version of WordPress packages. This is automated with the [Release Tool](#) that handles releases for the Gutenberg plugin. A successful RC1 release triggers the npm publishing job, and this needs to be approved by a Gutenberg Core team member. Locate the [“Build Gutenberg Plugin Zip” workflow](#) for the new version, and have it [approved](#).

We deliberately update the `wp/latest` branch within the Gutenberg repo with the content from the Gutenberg release `release/X.Y` (example `release/12.7`) branch at the time of the Gutenberg RC1 release. This is done to ensure that the `wp/latest` branch is as close as possible to the latest version of the Gutenberg plugin. It also practically removes the chances of conflicts while backporting to `trunk` commits with updates applied during publishing to `package.json` and `CHANGELOG.md` files. In the past, we had many issues in that aspect when doing npm publishing after the regular Gutenberg release a week later. When publishing the new package versions to npm, we pick at least the `minor` version bump to account for future bugfix or security releases.

Behind the scenes, all steps are automated via `./bin/plugin/cli.js npm-latest` command. For the record, the manual process would look very close to the following steps:

1. Ensure the WordPress `trunk` branch is open for enhancements.
2. Get the last published Gutenberg release branch with `git fetch`.
3. Check out the `wp/latest` branch.
4. Remove all files from the current branch: `git rm -r ..`

5. Check out all the files from the release branch: `git checkout release/x.x -- ..`
6. Commit all changes to the `wp/latest` branch with `git commit -m "Merge changes published in the Gutenberg plugin vX.X release"` and push to the repository.
7. Update the `CHANGELOG.md` files of the packages with the new publish version calculated and commit to the `wp/latest` branch. Assuming the package versions are written using this format `major.minor.patch`, make sure to bump at least the `minor` version bumps gets applied. For example, if the `CHANGELOG` of the package to be released indicates that the next unreleased version is `5.6.1`, choose `5.7.0` as a version in case of `minor` version. This is important as the patch version numbers should be reserved in case bug fixes are needed for a minor WordPress release (see below).
8. Log-in to `npm` via the console: `npm login`. Note that you should have 2FA enabled.
9. From the `wp/latest` branch, install `npm` dependencies with `npm ci`.
10. Run the script `npx lerna publish --no-private`.
 - When asked for the version numbers to choose for each package pick the values of the updated `CHANGELOG` files.
 - You'll be asked for your One-Time Password (OTP) a couple of times. This is the code from the 2FA authenticator app you use. Depending on how many packages are to be released you may be asked for more than one OTP, as they tend to expire before all packages are released.
 - If the publishing process ends up incomplete (perhaps because it timed-out or an bad OTP was introduced) you can resume it via [`npx lerna publish from-package`](#).
11. Finally, now that the `npm` packages are published, cherry-pick the commits created by `lerna` (“Publish” and the `CHANGELOG` update) into the `trunk` branch of Gutenberg.

[WordPress releases](#)

The following workflow is needed when bug or security fixes need to be backported into WordPress Core. This can happen in a few use-cases:

- During the `beta` and `RC` periods of the WordPress release cycle when `wp/X.Y` (example `wp/5.7`) branch for the release is already present.
 - For WordPress minor releases and WordPress security releases (example `5.1.1`).
1. Check out the relevant WordPress major branch (If the minor release is `5.2.1`, check out `wp/5.2`).
 2. Create a feature branch from that branch, and cherry-pick the merge commits for the needed bug fixes onto it. The cherry-picking process can be automated with the [`npm run other:cherry-pick` script](#).
 3. Create a Pull Request from this branch targeting the WordPress major branch used above.
 4. Merge the Pull Request using the “Rebase and Merge” button to keep the history of the commits.

Now, the `wp/X.Y` branch is ready for publishing `npm` packages. In order to start the process, go to Gutenberg’s GitHub repository’s Actions tab, and locate the [`“Publish npm packages” action`](#). Note the blue banner that says “This workflow has a `workflow_dispatch` event trigger.”, and expand the “Run workflow” dropdown on its right hand side.

Event ▾

Status ▾

Branch ▾

Action

Run workflow

Use workflow from

Branch: trunk ▾

Release type *

wp

WordPress major version (`wp` only)

6.0

Run workflow

Lish-npm-packages...

4 months ago

Failure

To publish packages to npm for the WordPress major release, select `trunk` as the branch to run the workflow from (this means that the script used to run the workflow comes from the `trunk` branch, though the packages themselves will be published from the `release` branch as long as the correct “Release type” is selected below), then select `wp` from the “Release type” dropdown and enter `X.Y` (example `5.2`) in the “WordPress major release” input field. Finally, press the green

“Run workflow” button. It triggers the npm publishing job, and this needs to be approved by a Gutenberg Core team member. Locate the “[Publish npm packages](#)” action for the current publishing, and have it [approved](#).

For the record, the manual process would look like the following:

1. Check out the WordPress branch used before (example wp/5.2).
2. `git pull`.
3. Run the `npx lerna publish patch --no-private --dist-tag wp-5.2` command (see more in [package release process](#)) but when asked for the version numbers to choose for each package, (assuming the package versions are written using this format `major.minor.patch`) make sure to bump only the `patch` version number. For example, if the last published package version for this WordPress branch was `5.6.0`, choose `5.6.1` as a version.

Note: For WordPress 5.0 and WordPress 5.1, a different release process was used. This means that when choosing npm package versions targeting these two releases, you won’t be able to use the next `patch` version number as it may have been already used. You should use the “metadata” modifier for these. For example, if the last published package version for this WordPress branch was `5.6.1`, choose `5.6.1+patch.1` as a version.

1. Optionally update the `CHANGELOG.md` files of the published packages with the new released versions and commit to the corresponding branch (Example `wp/5.2`).
2. Cherry-pick the `CHANGELOG` update commits, if any, into the `trunk` branch of Gutenberg.

Now, the npm packages should be ready and a patch can be created and committed into the corresponding WordPress SVN branch.

[**Standalone bugfix package releases**](#)

The following workflow is needed when packages require bug fixes or security releases to be published to `npm` outside of a regular release cycle.

Note: Both the `trunk` and `wp/latest` branches are restricted and can only be *pushed* to by the Gutenberg Core team.

Identify the commit hashes from the pull requests that need to be ported from the repo `trunk` branch to `wp/latest`

The `wp/latest` branch now needs to be prepared to release and publish the packages to `npm`.

Open a terminal and perform the following steps:

1. `git checkout trunk`
2. `git pull`
3. `git checkout wp/latest`
4. `git pull`

Before porting commits check that the `wp/latest` branch does not have any outstanding packages waiting to be published:

1. `git checkout wp/latest`
2. `npx lerna updated`

Now *cherry-pick* the commits from `trunk` to `wp/latest`, use `-m 1 commithash` if the commit was a pull request merge commit:

```
1.git cherry-pick -m 1 cb150a2  
2.git push
```

Whilst waiting for the GitHub actions build for `wp/latest`[branch to pass](#), identify and begin updating the `CHANGELOG.md` files:

```
1.git checkout wp/latest  
2.npx lerna updated  
  > Example  
  >  
  > shell  
  > npx lerna updated  
  > @wordpress/e2e-tests  
  > @wordpress/jest-preset-default  
  > @wordpress/scripts  
  > lerna success found 3 packages ready to publish  
  >
```

Check the versions listed in the current `CHANGELOG.md` file, looking through the commit history of a package e.g [@wordpress/scripts](#) and look out for “*chore(release): publish*” and “*Update changelogs*” commits to determine recent version bumps, then looking at the commits since the most recent release should aid with discovering what changes have occurred since the last release.

Note: You may discover the current version of each package is not up to date, if so updating the previously released versions would be appreciated.

Now, the `wp/latest` branch is ready for publishing npm packages. In order to start the process, go to Gutenberg’s GitHub repository’s Actions tab, and locate the [“Publish npm packages” action](#). Note the blue banner that says “This workflow has a `workflow_dispatch` event trigger.”, and expand the “Run workflow” dropdown on its right hand side.

Event ▾

Status ▾

Branch ▾

Action

Run workflow

Use workflow from

Branch: trunk ▾

Release type *

wp

WordPress major version (`wp` only)

6.0

Run workflow

Lish-npm-packages...

4 months ago

Failure

To publish packages to npm with bugfixes, select bug fix from the “Release type” dropdown and leave empty “WordPress major release” input field. Finally, press the green “Run workflow” button. It triggers the npm publishing job, and this needs to be approved by a Gutenberg Core team member. Locate the [“Publish npm packages” action](#) for the current publishing, and have it [approved](#).

Behind the scenes, the rest of the process is automated with `./bin/plugin/cli.js npm-fix` command. For the record, the manual process would look very close to the following steps:

1. Check out the `wp/latest` branch.
2. Update the `CHANGELOG.md` files of the packages with the new publish version calculated and commit to the `wp/latest` branch.
3. Log-in to npm via the console: `npm login`. Note that you should have 2FA enabled.
4. From the `wp/latest` branch, install npm dependencies with `npm ci`.
5. Run the script `npx lerna publish --no-private`.
 - When asked for the version numbers to choose for each package pick the values of the updated `CHANGELOG` files.
 - You'll be asked for your One-Time Password (OTP) a couple of times. This is the code from the 2FA authenticator app you use. Depending on how many packages are to be released you may be asked for more than one OTP, as they tend to expire before all packages are released.
 - If the publishing process ends up incomplete (perhaps because it timed-out or an bad OTP was introduced) you can resume it via [`npx lerna publish from-package`](#).
6. Finally, now that the npm packages are published, cherry-pick the commits created by lerna (“Publish” and the `CHANGELOG` update) into the `trunk` branch of Gutenberg.

Development releases

As noted in the [Synchronizing Gutenberg Plugin](#) section, packages publishing happens every two weeks from the `wp/latest` branch. It's also possible to use the development release to test the upcoming changes present in the `trunk` branch at any time. We are taking advantage of [package distribution tags](#) that make it possible to consume the future version of the codebase according to npm guidelines:

By default, the `latest` tag is used by npm to identify the current version of a package, and `npm install <pkg>` (without any `@<version>` or `@<tag>` specifier) installs the `latest` tag. Typically, projects only use the `latest` tag for stable release versions, and use other tags for unstable versions such as prereleases.

In our case, we use the `next` distribution tag for code. Developers that want to install such a version of the package need to type:

```
npm install @wordpress/components@next
```

In order to start the publishing process for development version of npm packages, go to Gutenberg's GitHub repository's Actions tab, and locate the “[Publish npm packages](#)” action. Note the blue banner that says “This workflow has a `workflow_dispatch` event trigger.”, and expand the “Run workflow” dropdown on its right hand side.

Event ▾

Status ▾

Branch ▾

Action

Run workflow

Use workflow from

Branch: trunk ▾

Release type *

wp

WordPress major version (`wp` only)

6.0

Run workflow

Lish-npm-packages...

4 months ago

Failure

To publish development packages to npm, select development from the “Release type” dropdown and leave empty “WordPress major release” input field. Finally, press the green “Run workflow” button. It triggers the npm publishing job, and this needs to be approved by a Gutenberg Core team member. Locate the [“Publish npm packages” action](#) for the current publishing, and have it [approved](#).

Behind the scenes, the release process is fully automated via `./bin/plugin/cli.js npm-next` command. It ensures the `wp/next` branch is synchronized with the latest release branch (`release/X.Y`) created for the Gutenberg plugin. To avoid collisions in the versioning of packages, we always include the newest commit's sha, for example, `@wordpress/block-editor@5.2.10-next.645224df70.0`.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Gutenberg Release Process”](#)

[Previous Managing Packages](#) [Previous: Managing Packages](#)
[Next Cherry-picking automation](#) [Next: Cherry-picking automation](#)

Cherry-picking automation

In this article

Table of Contents

- [Can I use a different label than Backport to WP Beta/RC?](#)
- [How can I use it for a Gutenberg plugin release?](#)
- [Future improvements](#)

[↑ Back to top](#)

`npm run other:cherry-pick` automates cherry-picking Pull Requests with a specific label into the **current branch**.

It's especially useful for major WordPress releases as by default the script looks for merged Pull Requests with the `Backport to WP Beta/RC` label.

You can also use it in different scenarios by passing a custom label as the first argument. See the Gutenberg plugin release example at the end of this document.

Running `npm run other:cherry-pick` yields the following prompt:

You are on branch "wp/6.2".

This script will:

- Cherry-pick the merged PRs labeled as "Backport to WP Beta/RC" to this branch
- Ask whether you want to push this branch
- Comment on each PR
- Remove the label from each PR

The last two actions will be performed USING YOUR GITHUB ACCOUNT that you linked to your GitHub CLI (gh command)

Do you want to proceed? (Y/n)

Here's what happens once you agree:

Trying to cherry-pick one by one..

```
$ git pull origin wp/6.2 --rebase...
$ git fetch origin trunk...
```

Found the following PRs to cherry-pick:

#41198 - Site Editor: Set min-width for styles preview

Fetching commit IDs... Done!

#41198 - 860a39665c318d33027d - Site Editor: Set min-width for...

Trying to cherry-pick one by one...

Cherry-picking round 1:

cherry-pick commit: afe9b757b4 for PR: #41198 - Site Editor: Set min-Cherry-picking finished!

Summary:

1 PRs got cherry-picked cleanly
0 PRs failed

About to push to origin/wp/6.2

Do you want to proceed? (Y/n)

This run was successful, yay! You can use this moment to confirm the correct PRs were cherry-picked.

What if the cherry-picks didn't apply cleanly? The script would apply the rest and retry.

If some cherry-picks still failed, the script would skip them and let you know which conflicts require a manual resolution.

Either way, here's what happens once you proceed past the cherry-picking stage:

Pushing to origin/wp/6.2

Commenting and removing labels...

41198: I just cherry-picked this PR to the wp/6.2 branch to get it included!
Done!

The commenting part is optional and only possible if you have the [gh console utility](#) installed.

Can I use a different label than Backport to WP Beta/RC?

Yes! Pass it as the first argument:

```
npm run other:cherry-pick "Another Label"
```

How can I use it for a Gutenberg plugin release?

```
# Switch to the release branch  
git checkout release/X.Y  
  
# Cherry-pick all the merged PRs with a relevant backport label  
npm run other:cherry-pick "Backport to Gutenberg RC"
```

Future improvements

In the future, it would be great if the script automatically selected the relevant label based on the currently selected branch:

- release/X.Y – plugin release – “Backport to Gutenberg RC”
- wp/X.Y – WP release – “Backport to WP Beta/RC”

First published

August 4, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Cherry-picking automation](#)

[Previous Gutenberg Release Process](#) [Previous: Gutenberg Release Process](#)
[Next React Native mobile editor](#) [Next: React Native mobile editor](#)

React Native mobile editor

In this article

Table of Contents

- [Mind the mobile](#)
- [Native mobile specific files](#)
- [Running Gutenberg Mobile on Android and iOS](#)
- [Native mobile E2E tests in Continuous Integration](#)
- [Debugging the native mobile unit tests](#)
- [Internationalization \(i18n\)](#)

[↑ Back to top](#)

The Gutenberg repository includes the source for the [React Native](#) based editor for mobile.

Mind the mobile

Contributors need to ensure that they update any affected native mobile files during code refactorings because we cannot yet rely on automated tooling to do this for us. For example, renaming a function or a prop should also be performed in the native modules too, otherwise, the mobile client will break. We have added some mobile specific CI tests as safeguards in place in PRs, but we're still far from done. Please bear with us and thank you in advance. ❤️👨‍💻

Native mobile specific files

The majority of the code shared with native mobile is in the very same JavaScript module and SASS style files. In the cases where the code paths need to diverge, a `.native.js` or `.native.scss` variant of the file is created. In some cases, platform specific files can be also found for Android (`.android.js`) or iOS (`.ios.js`).

Running Gutenberg Mobile on Android and iOS

For instructions on how to run the **Gutenberg Mobile Demo App** on Android or iOS, see [Getting Started for the React Native based Mobile Gutenberg](#)

Also, the mobile client is packaged and released via the [official WordPress apps](#). Even though the build pipeline is slightly different than the mobile demo apps and lives in its own repo for now ([here's the native mobile repo](#)), the source code itself is taken directly from this repo and the “web” side codepaths.

Native mobile E2E tests in Continuous Integration

If you encounter a failed Android/iOS test on your pull request, we recommend the following steps:

1. Re-running the failed GitHub Action job ([guide for how to re-run](#)) – This should fix failed tests the majority of the time.
2. You can check if the test is failing locally by following the steps to run the E2E test on your machine from the [E2E testing documentation](#).
3. In addition to reading the logs from the E2E test, you can download a video recording from the Artifacts section of the GitHub job that may have additional useful information.
4. Check if any changes in your PR would require corresponding changes to `.native.js` versions of files.
5. Lastly, if you’re stuck on a failing mobile test, feel free to reach out to contributors on Slack in the #mobile or #core-editor chats in the WordPress Core Slack, [free to join](#).

Debugging the native mobile unit tests

Follow the instructions in [Native mobile testing](#) to locally debug the native mobile unit tests when needed.

Internationalization (i18n)

Further information about this topic can be found in the [React Native Internationalization Guide](#).

First published

December 7, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: React Native mobile editor”](#)

[Previous Cherry-picking automation](#) [Previous: Cherry-picking automation](#)

[Next Getting Started for the React Native based Mobile Gutenberg](#) [Next: Getting Started for the React Native based Mobile Gutenberg](#)

Getting Started for the React Native based Mobile Gutenberg

In this article

[Table of Contents](#)

- [Prerequisites](#)
- [Clone the project](#)
- [Set up](#)
- [Run](#)
 - [Running on other iOS device simulators](#)
 - [Customizing the demo Editor](#)
 - [Troubleshooting](#)
- [Developing with Visual Studio Code](#)
- [Unit tests](#)
- [Writing and running unit tests](#)
- [End-to-end tests](#)

[↑ Back to top](#)

Welcome! This is the Getting Started guide for the native mobile port of the block editor, targeting Android and iOS devices. Overall, it's a React Native library to be used in parent greenfield or brownfield apps. Continue reading for information on how to build, test, and run it.

[Prerequisites](#)

For a developer experience closer to the one the project maintainers current have, make sure you have the following tools installed:

- git
- [nvm](#)
- Node.js and npm (use nvm to install them)
- [Android Studio](#) to be able to compile the Android version of the app

- [Xcode](#) to be able to compile the iOS app
- CocoaPods (`sudo gem install cocoapods`) needed to fetch React and third-party dependencies.

Note that the OS platform used by the maintainers is macOS but the tools and setup should be usable in other platforms too.

Clone the project

```
git clone https://github.com/WordPress/gutenberg.git
```

Set up

Note that the commands described here should be run in the top-level directory of the cloned project. Before running the demo app, you need to download and install the project dependencies. This is done via the following command:

```
nvm install  
npm ci  
npm run native preios
```

Run

```
npm run native start:reset
```

Runs the packager (Metro) in development mode. The packager stays running to serve the app bundle to the clients that request it.

With the packager running, open another terminal window and use the following command to compile and run the Android app:

```
npm run native android
```

The app should now open in a connected device or a running emulator and fetch the JavaScript code from the running packager.

To compile and run the iOS variant of the app using the *default* simulator device, use:

```
npm run native ios
```

which will attempt to open your app in the iOS Simulator if you're on a Mac and have it installed.

Running on other iOS device simulators

To compile and run the app using a different device simulator, use the following, noting the double sets of `--` to pass the simulator option down to the `react-native` CLI.

```
npm run native ios -- -- --simulator="DEVICE_NAME"
```

For example, if you'd like to run in an iPhone Xs Max, try:

```
npm run native ios -- -- --simulator="iPhone Xs Max"
```

To see a list of all of your available iOS devices, use `xcrun simctl list devices`.

Customizing the demo Editor

By default, the Demo editor renders most of the supported core blocks. This is helpful to showcase the editor's capabilities, but can be distracting when focusing on a specific block or feature. One can customize the editor's initial state by leveraging the `native.block_editor_props` hook in a `packages/react-native-editor/src/setup-local.js` file.

Example setup-local.js

```
/***
 * WordPress dependencies
 */
import { addFilter } from '@wordpress/hooks';

export default () => {
    addFilter(
        'native.block_editor_props',
        'core/react-native-editor',
        ( props ) => {
            return {
                ...props,
                initialHtml,
            };
        }
    );
};

const initialHtml = `
<!-- wp:heading -->
<h2 class="wp-block-heading">Just a Heading</h2>
<!-- /wp:heading -->
`;
```

Troubleshooting

If the Android emulator doesn't start correctly, or compiling fails with `Could not initialize class org.codehaus.groovy.runtime.InvokerHelper` or similar, it may help to double check the set up of your development environment against the latest requirements in [React Native's documentation](#). With Android Studio, for example, you will need to configure the `ANDROID_HOME` environment variable and ensure that your version of JDK matches the latest requirements.

Some times, and especially when tweaking anything in the `package.json`, Babel configuration (`.babelrc`) or the Jest configuration (`jest.config.js`), your changes might seem to not take effect as expected. On those times, you might need to stop the metro bundler process and restart it with `npm run native start:reset`. Other times, you might want to reinstall the NPM packages from scratch and the `npm run native clean:install` script can be handy.

Developing with Visual Studio Code

Although you're not required to use Visual Studio Code for developing gutenberg-mobile, it is the recommended IDE and we have some configuration for it.

When you first open the project in Visual Studio, you will be prompted to install some recommended extensions. This will help with some things like type checking and debugging.

One of the extensions we are using is the [React Native Tools](#). This allows you to run the packager from VSCode or launch the application on iOS or Android. It also adds some debug configurations so you can set breakpoints and debug the application directly from VSCode. Take a look at the [extension documentation](#) for more details.

Unit tests

Use the following command to run the test suite:

```
npm run test:native
```

It will run the [jest](#) test runner on your tests. The tests are running on the desktop against Node.js.

To run the tests with debugger support, start it with the following CLI command:

```
npm run test:native:debug
```

Then, open `chrome://inspect` in Chrome to attach the debugger (look into the “Remote Target” section). While testing/developing, feel free to sprinkle `debugger` statements anywhere in the code that you’d like the debugger to break.

Writing and running unit tests

This project is set up to use [jest](#) for tests. You can configure whatever testing strategy you like, but jest works out of the box. Create test files in directories called `__tests__` or with the `.test.js` extension to have the files loaded by jest. See an example test [here](#). The [jest documentation](#) is also a wonderful resource, as is the [React Native testing tutorial](#).

End-to-end tests

In addition to unit tests, the Mobile Gutenberg (MG) project relies upon end-to-end (E2E) tests to automate testing critical flows in an environment similar to that of an end user. We generally prefer unit tests due to their speed and ease of maintenance. However, assertions that require OS-level features (e.g. complex gestures, text selection) or visual regression testing (e.g. dark mode, contrast levels) we use E2E tests.

The E2E tests are found in the [packages/react-native-editor/_device-tests_](#) directory. Additional documentation on running and contributing to these tests can be found in the [tests directory](#).

First published

December 7, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Getting Started for the React Native based Mobile Gutenberg”](#)

[Previous React Native mobile editor](#) [Previous: React Native mobile editor](#)

[Next Setup guide for React Native development \(macOS\)](#) [Next: Setup guide for React Native development \(macOS\)](#)

Setup guide for React Native development (macOS)

In this article

Table of Contents

- [Clone Gutenberg](#)
 - [Install node and npm](#)
 - [Do you have an older existing Gutenberg checkout?](#)
- [iOS](#)
 - [CocoaPods](#)
 - [Set up Xcode](#)
 - [react-native doctor](#)
 - [Run the demo app](#)
- [Android](#)
 - [Java Development Kit \(JDK\)](#)
 - [Set up Android Studio](#)
 - [Update Paths](#)
 - [Create a new device image](#)
 - [Run the demo app](#)
- [Unit Tests](#)
- [Integration Tests](#)
 - [iOS Integration Tests](#)
 - [Android Integration Tests](#)

[↑ Back to top](#)

Are you interested in contributing to the native mobile editor? This guide is a detailed walk through designed to get you up and running!

Note that these instructions are primarily focused on the macOS environment. For other environments, [the React Native quickstart documentation](#) has helpful pointers and steps for getting set up.

[**Clone Gutenberg**](#)

```
git clone git@github.com:WordPress/gutenberg.git
```

Install node and npm

If you're working in multiple JS projects, a node version manager may make sense. A manager will let you switch between different node and npm versions of your choosing.

We recommend [nvm](#).

After installing nvm, run the following from the top-level directory of the cloned project:

```
nvm install 'lts/*'  
nvm alias default 'lts/*' # sets this as the default when opening a new terminal  
nvm use # switches to the project settings
```

Then install dependencies:

```
npm ci
```

Do you have an older existing Gutenberg checkout?

If you have an existing Gutenberg checkout be sure to fully clean `node_modules` and re-install dependencies.

This may help avoid errors in the future.

```
npm run distclean  
npm ci
```

iOS

CocoaPods

[CocoaPods](#) is required to fetch React and third-party dependencies. The steps to install it vary depending on how Ruby is managed on your machine.

System Ruby

If you're using the default Ruby available with MacOS, you'll need to use the `sudo` command to install gems like Cocoapods:

```
sudo gem install cocoapods
```

Note, Mac M1 is not directly compatible with Cocoapods. If you encounter issues, try running these commands to install the `ffi` package, which will enable pods to be installed with the proper architecture:

```
sudo arch -x86_64 gem install ffi  
arch -x86_64 pod install
```

Ruby Manager

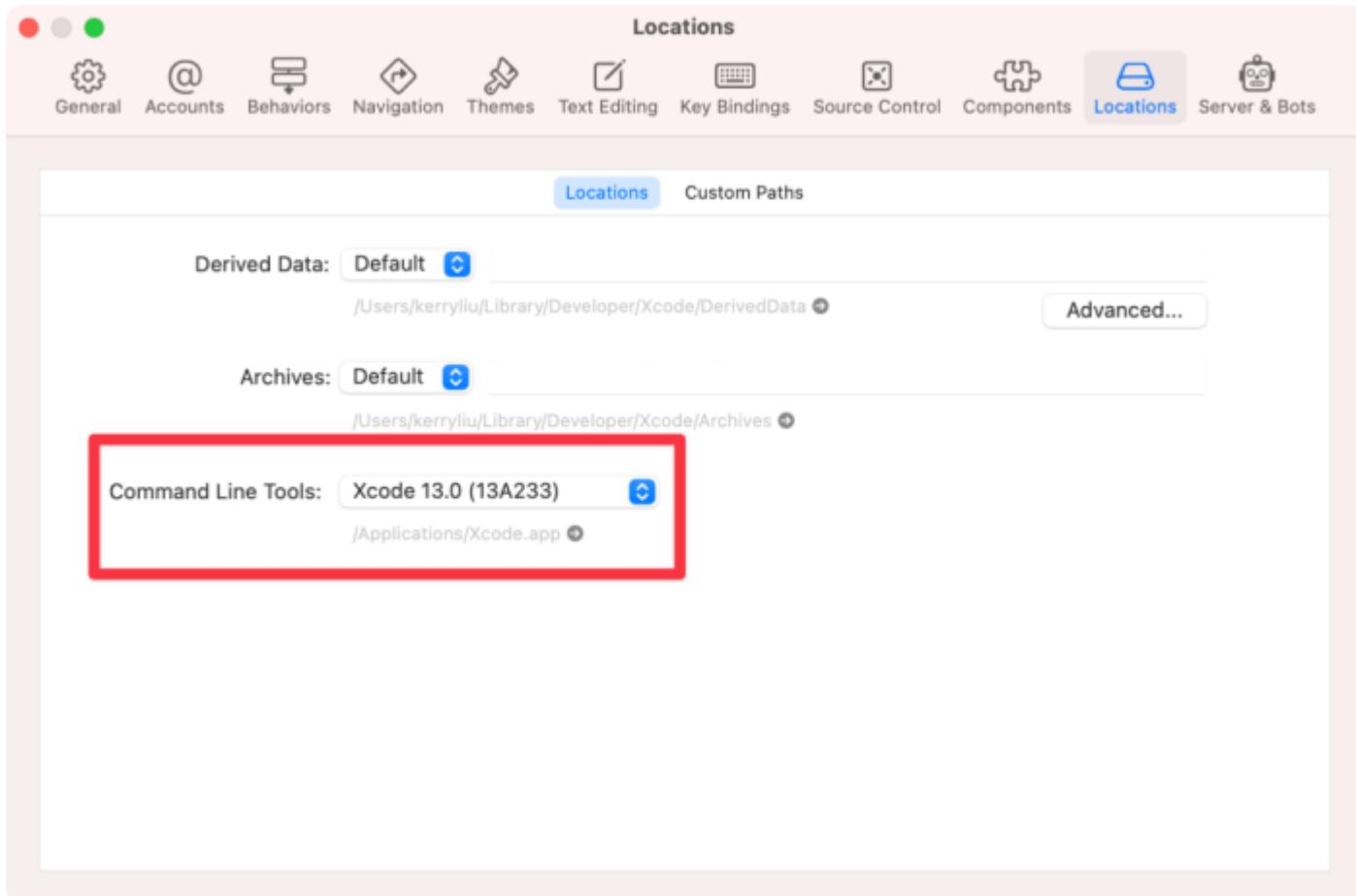
It may not be necessary to manually install Cocoapods or the `ffi` package if you're using a Ruby Version manager. Please refer to your chosen manager's documentation for guidance.

[rbenv](#) is the recommended manager if you're running Gutenberg from within [the WordPress iOS app](#) (vs. only the demo app).

Set up Xcode

Install [Xcode](#) via the app store and then open it up:

- Accept the license agreement.
- Verify that Xcode > Preferences > Locations > Command Line Tools points to the current Xcode version.



react-native doctor

[react-native doctor](#) can be used to identify anything that's missing from your development environment. From your Gutenberg checkout, or relative to `/packages/react-native-editor` folder, run:

```
npx @react-native-community/cli doctor
```

```
● ● ●
Time:      59.168 s
Ran all test suites.

gutenberg on ✚ trunk [$] took 1m5s
❯ npx @react-native-community/cli doctor
npx: installed 573 in 12.732s
Common
  ✓ Node.js
  ✓ npm
  ✗ Watchman - Used for watching changes in the filesystem when in development mode
    - Version found: N/A
    - Version supported: 4.x

Android
  ✗ JDK
    - Version found: N/A
    - Version supported: 1.8.x || ≥ 9
  ✗ Android Studio - Required for building and installing your app on Android
  ✗ Android SDK - Required for building and installing your app on Android
    - Versions found: N/A
    - Version supported: Not Found
  ✗ ANDROID_HOME

iOS
  ✗ Xcode - Required for building and installing your app on iOS
    - Version found: N/A
    - Version supported: ≥ 10.x
  ✗ CocoaPods - Required for installing iOS dependencies
  ● ios-deploy - Required for installing your app on a physical device with the CLI

Errors:  7
Warnings: 1

Usage
  › Press f to try to fix issues.
  › Press e to try to fix errors.
  › Press w to try to fix warnings.
  › Press Enter to exit.
```

See if `doctor` can fix both “common” and “iOS” issues. (Don’t worry if “Android” still has ✗ at this stage, we’ll get to those later!)

Run the demo app

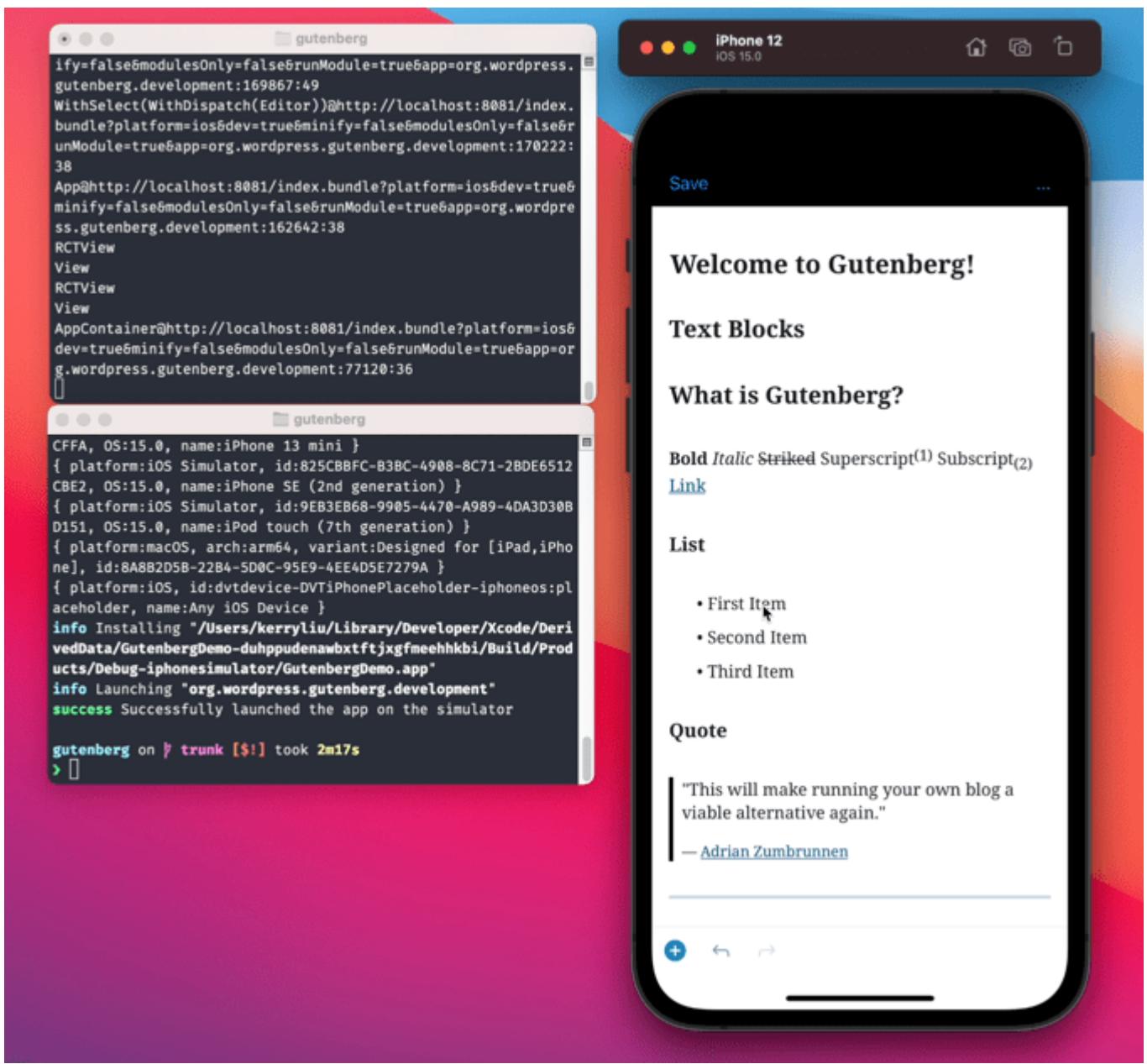
Once all common and iOS issues are resolved, try:

```
npm run native start:reset #starts metro
```

In another terminal type:

```
npm run native ios
```

After waiting for everything to build, the demo app should be running from the iOS simulator:



Android

Java Development Kit (JDK)

The JDK recommended in [the React Native documentation](#) is called Azul Zulu. It can be installed using [Homebrew](#). To install it, run the following commands in a terminal after installing Homebrew:

```
brew tap homebrew/cask-versions  
brew install --cask zulu11
```

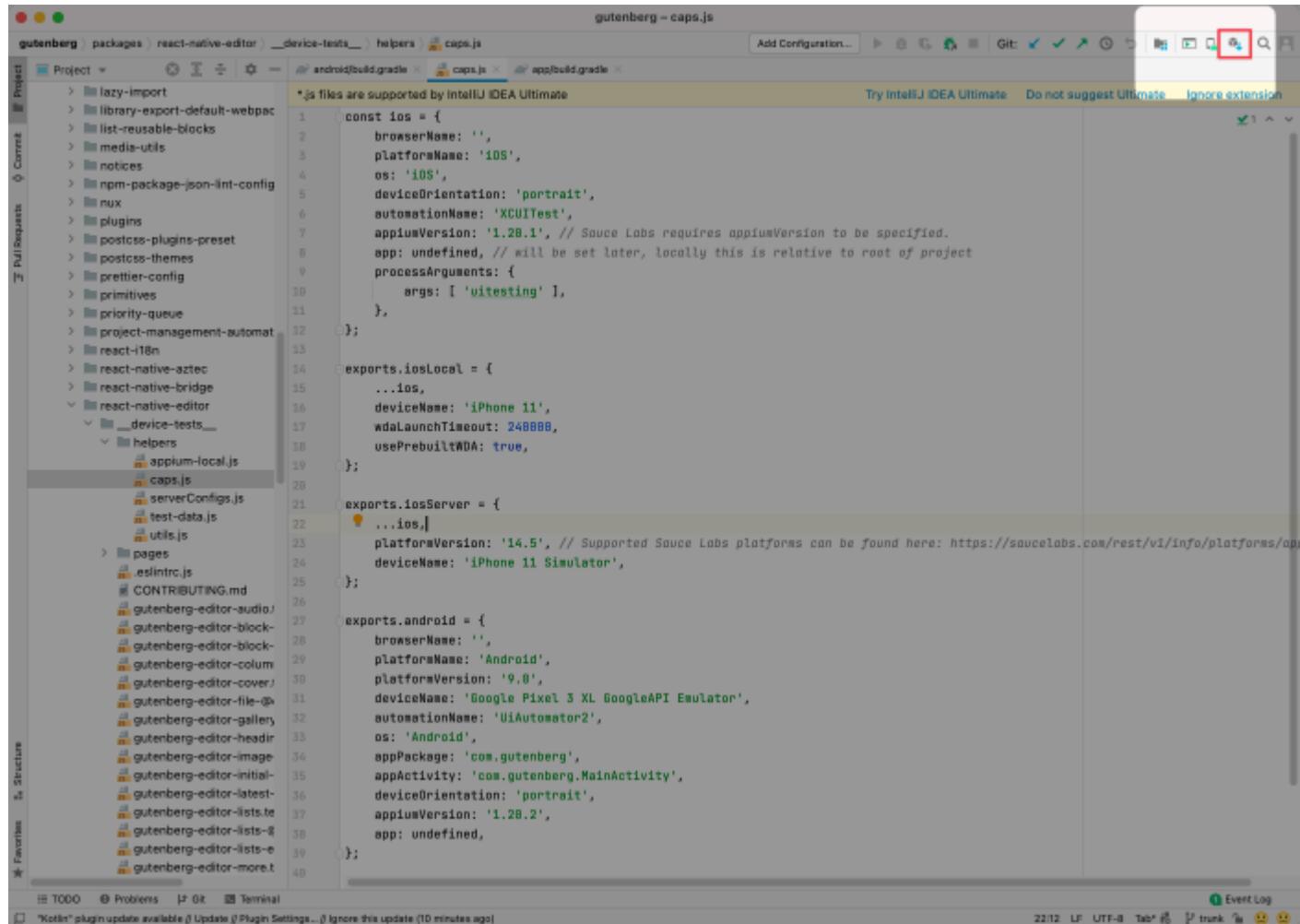
If you already have a JDK installed on your system, it should be JDK 11 or newer.

Set up Android Studio

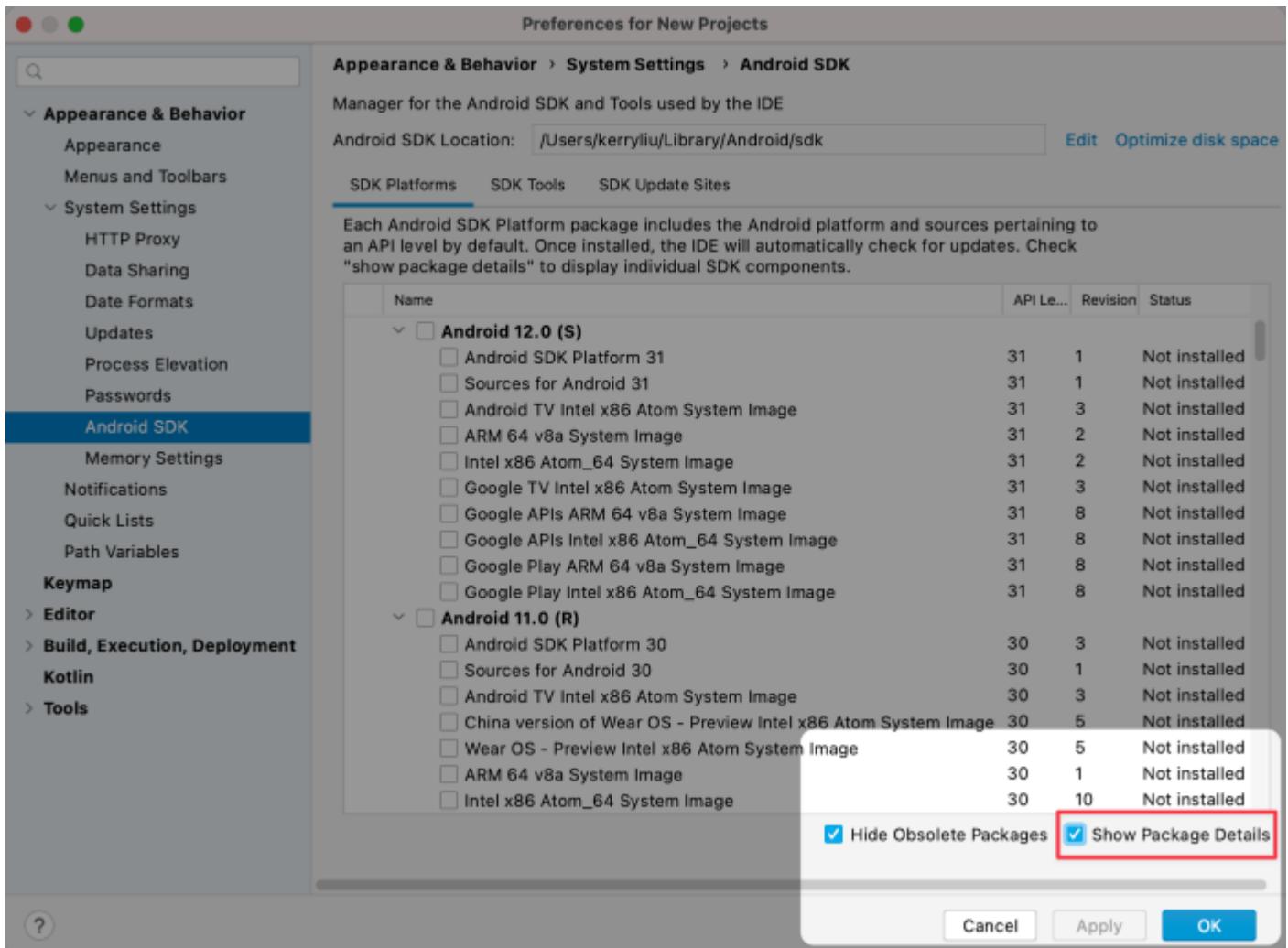
To compile the Android app, [download Android Studio](#).

Next, open an existing project and select the Gutenberg folder you cloned.

From here, click on the cube icon that's highlighted in the following screenshot to access the SDK Manager. Another way to the SDK Manager is to navigate to Tools > SDK Manager:



We can download SDK platforms, packages and other tools on this screen. Specific versions are hidden behind the “Show package details” checkbox, check it, since our build requires specific versions for E2E and development:



Check all related packages from [build.gradle](#). Then click on “Apply” to download items. There may be other related dependencies from build.gradle files in node_modules.

If you don't want to dig through files, stack traces will complain of missing packages, but it does take quite a number of tries if you go through this route.

gutenberg - android/build.gradle

```
buildscript {
    ext {
        gradlePluginVersion = '4.2.2'
        kotlinVersion = '1.5.20'
        buildToolsVersion = "29.0.3" ←
        minSdkVersion = 21
        compileSdkVersion = 29 ←
        targetSdkVersion = 29 ←
        supportLibVersion = '28.0.0'
        wordpressUtilsVersion = '1.22'
        ndkVersion = "28.1.5948944" ←
    }
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath "com.android.tools.build:gradle:$gradlePluginVersion"
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlinVersion"
    }
}

allprojects {
    repositories {
        mavenLocal()
        maven {
            // All of React Native (JS, Obj-C sources, Android binaries) is installed from npm
            url("$rootDir/../node_modules/react-native/android")
        }
        maven {
            url "https://a8c-libs.s3.amazonaws.com/android"
            content {
                includeGroup "org.wordpress"
                includeGroup "org.wordpress.sztec"
            }
        }
        maven { url "https://a8c-libs.s3.amazonaws.com/android/hermes-mirror" }
        maven { url 'https://jitpack.io' }
        google()
        // ...
    }
}
buildscript{...} + ext{...}
```

Project: gutenberg | packages | react-native-editor | android | build.gradle

git: ✓ ✓ ✓ ✓ ✓ ✓

File: build.gradle

Line: 1

Column: 1

Editor: IntelliJ IDEA

Preferences for New Projects

Appearance & Behavior > System Settings > Android SDK

Manager for the Android SDK and Tools used by the IDE

Android SDK Location: /Users/kerryliu/Library/Android/sdk [Edit](#) [Optimize disk space](#)

SDK Platforms SDK Tools SDK Update Sites

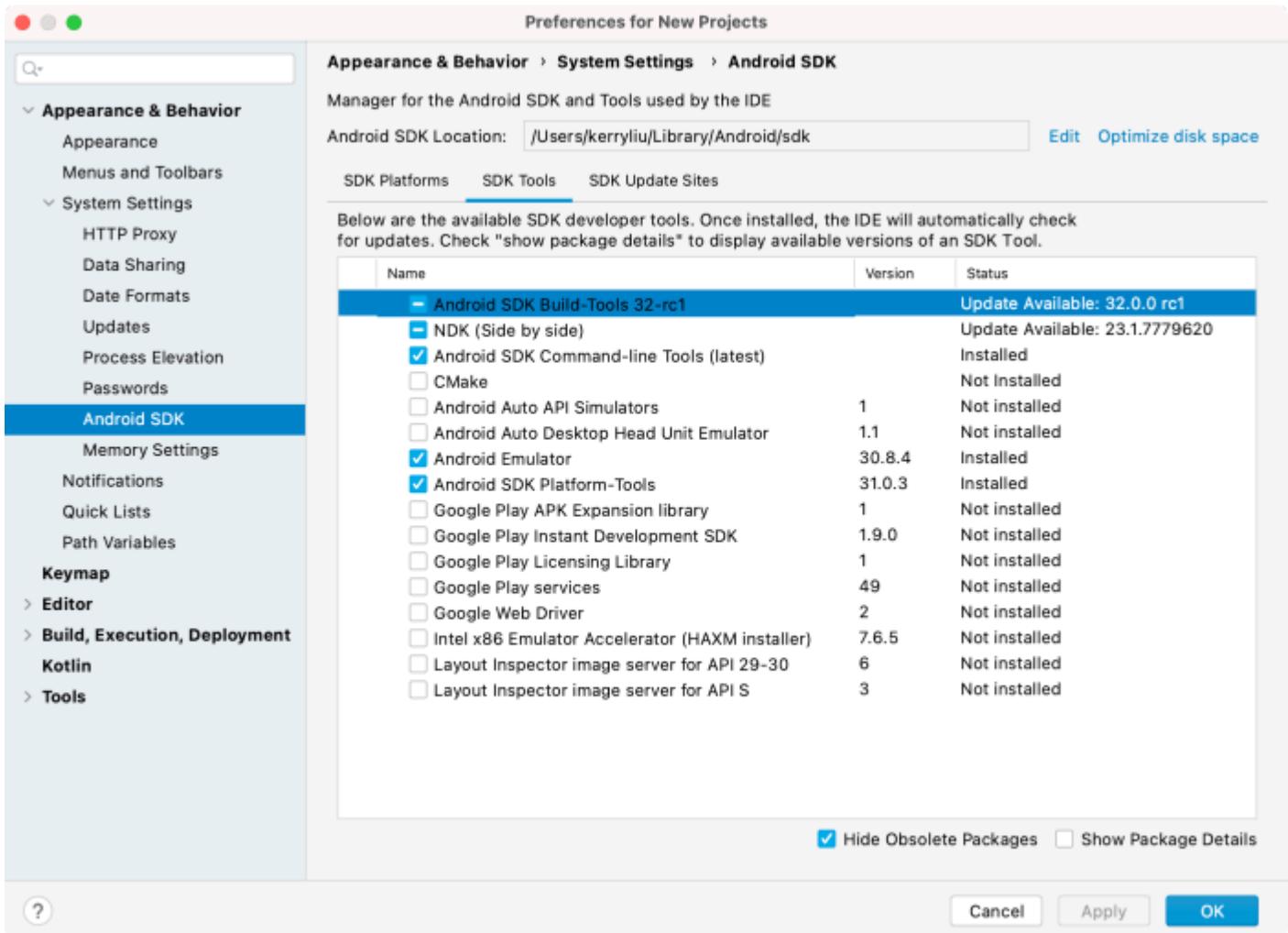
Each Android SDK Platform package includes the Android platform and sources pertaining to an API level by default. Once installed, the IDE will automatically check for updates. Check "show package details" to display individual SDK components.

Name	API Le...	Revision	Status
Android 10.0 (Q)			
Android SDK Platform 29	29	5	Installed
Sources for Android 29	29	1	Installed
Android TV Intel x86 Atom System Image	29	3	Not installed
Intel x86 Atom System Image	29	8	Not installed
Intel x86 Atom_64 System Image	29	8	Not installed
Google APIs ARM 64 v8a System Image	29	12	Installed
Google APIs Intel x86 Atom System Image	29	12	Not installed
Google APIs Intel x86 Atom_64 System Image	29	12	Not installed
Google Play ARM 64 v8a System Image	29	9	Installed
Google Play Intel x86 Atom System Image	29	8	Not installed
Google Play Intel x86 Atom_64 System Image	29	8	Not installed
Android 9.0 (Pie)			
Android SDK Platform 28	28	6	Not installed
Sources for Android 28	28	1	Not installed
Android TV Intel x86 Atom System Image	28	10	Not installed
China version of Wear OS Intel x86 Atom System Image	28	6	Not installed
Wear OS Intel x86 Atom System Image	28	6	Not installed
Intel x86 Atom System Image	28	4	Not installed
Intel x86 Atom_64 System Image	28	4	Not installed

Hide Obsolete Packages Show Package Details

?

[Cancel](#) [Apply](#) [OK](#)



Update Paths

Export the following env variables and update \$PATH. We can normally add this to our `~/.zshrc` file if we're using zsh
in our terminal, or `~/.bash_profile` if the terminal is still using bash.

```
### Java that comes with Android Studio:
export JAVA_HOME=/Applications/Android\ Studio.app/Contents/jre/Contents/Home
### Android Home is configurable in Android Studio. Go to Preferences > System
export ANDROID_HOME=$HOME/Library/Android/sdk
export PATH=$PATH:$ANDROID_HOME/emulator
export PATH=$PATH:$ANDROID_HOME/tools
export PATH=$PATH:$ANDROID_HOME/tools/bin
export PATH=$PATH:$ANDROID_HOME/platform-tools
```

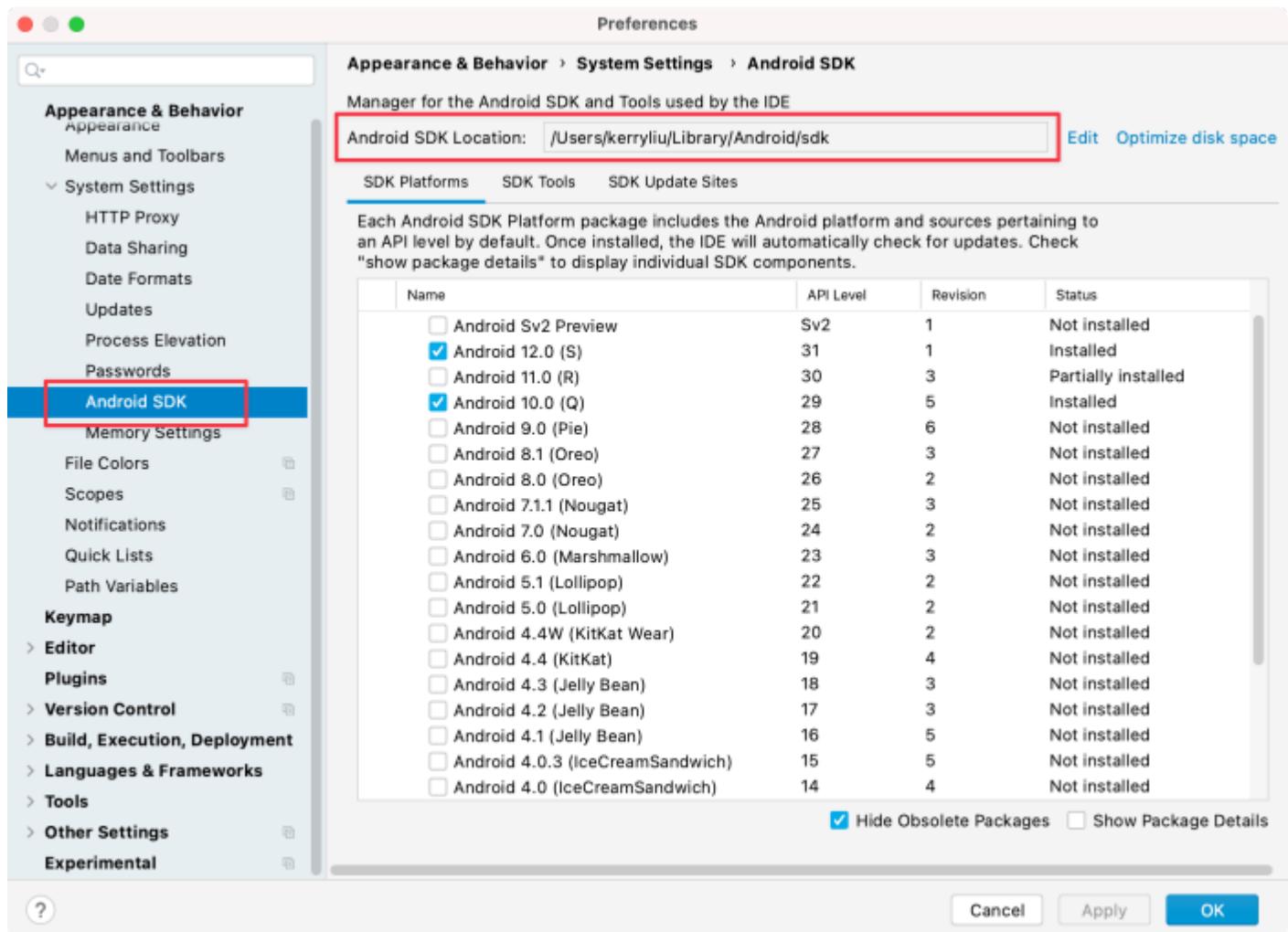
Save then source, or open a new terminal to pick up changes.

```
source ~/.zshrc
```

or

```
source ~/.bash_profile
```

If the SDK path can't be found, you can verify its location by visiting Android Studio > Preferences > System Settings > Android SDK



Create a new device image

Next, let's create a virtual device image. Click on the phone icon with an android to the bottom-right.

```
gutenberg - caps.js
gutenberg | packages | react-native-editor | __device-tests__ | helpers | caps.js
Project | Recent | Add Configuration... | Git: ✓ | Try IntelliJ IDEA Ultimate | Do not suggest Ultimate | Ignore extension
File | Settings | Help | View | Editor | Tools | Run | Build | Structure | Favorites | Event Log
22/12 LF UTF-8 Tab: 0 trunk /a/ ☺ ☹
```

```
*js files are supported by IntelliJ IDEA Ultimate
1 const ios = {
2   browserName: '',
3   platformName: 'iOS',
4   os: 'iOS',
5   deviceOrientation: 'portrait',
6   automationName: 'XCUITest',
7   appiumVersion: '1.28.1', // Sauce Labs requires appiumVersion to be specified.
8   app: undefined, // will be set later, locally this is relative to root of project
9   processArguments: [
10     args: [ 'uitesting' ],
11   ],
12 };
13
14 exports.iosLocal = {
15   ...ios,
16   deviceName: 'iPhone 11',
17   wdaLaunchTimeout: 240000,
18   usePrebuiltWDA: true,
19 };
20
21 exports.iosServer = {
22   ...ios,
23   platformVersion: '14.5', // Supported Sauce Labs platforms can be found here: https://saucelabs.com/rest/v1/info/platforms/op
24   deviceName: 'iPhone 11 Simulator',
25 };
26
27 exports.android = {
28   browserName: '',
29   platformName: 'Android',
30   platformVersion: '9.0',
31   deviceName: 'Google Pixel 3 XL GoogleAPI Emulator',
32   automationName: 'UiAutomator2',
33   os: 'Android',
34   appPackage: 'com.gutenberg',
35   appActivity: 'com.gutenberg.MainActivity',
36   deviceOrientation: 'portrait',
37   appiumVersion: '1.28.2',
38   app: undefined,
39 };
40
```

This brings up the “Android Virtual Device Manager” or (AVD). Click on “Create Virtual Device”. Pick a phone type of your choice:

Virtual Device Configuration

Select Hardware

Choose a device definition

Category	Name	Play Store	Size	Resolution	Density
TV	Pixel XL		5.5"	1440x...	560dpi
Phone	Pixel 5		6.0"	1080x...	440dpi
Wear OS	Pixel 4a		5.8"	1080x...	440dpi
Tablet	Pixel 4 XL		6.3"	1440x...	560dpi
Automotive	Pixel 4	►	5.7"	1080x...	440dpi
	Pixel 3a XL		6.0"	1080x...	400dpi
	Pixel 3a	►	5.6"	1080x...	440dpi
	Pixel 3 XL		6.3"	1440x...	560dpi
	Pixel 3	►	5.46"	1080x...	440dpi
	Pixel 2 XL		5.99"	1440x...	560dpi
	Pixel 2	►	5.0"	1080x...	420dpi

[New Hardware Profile](#) [Import Hardware Profiles](#) [Clone Device...](#)

Pixel 5

Size: large
Ratio: long
Density: 440dpi

6.0"

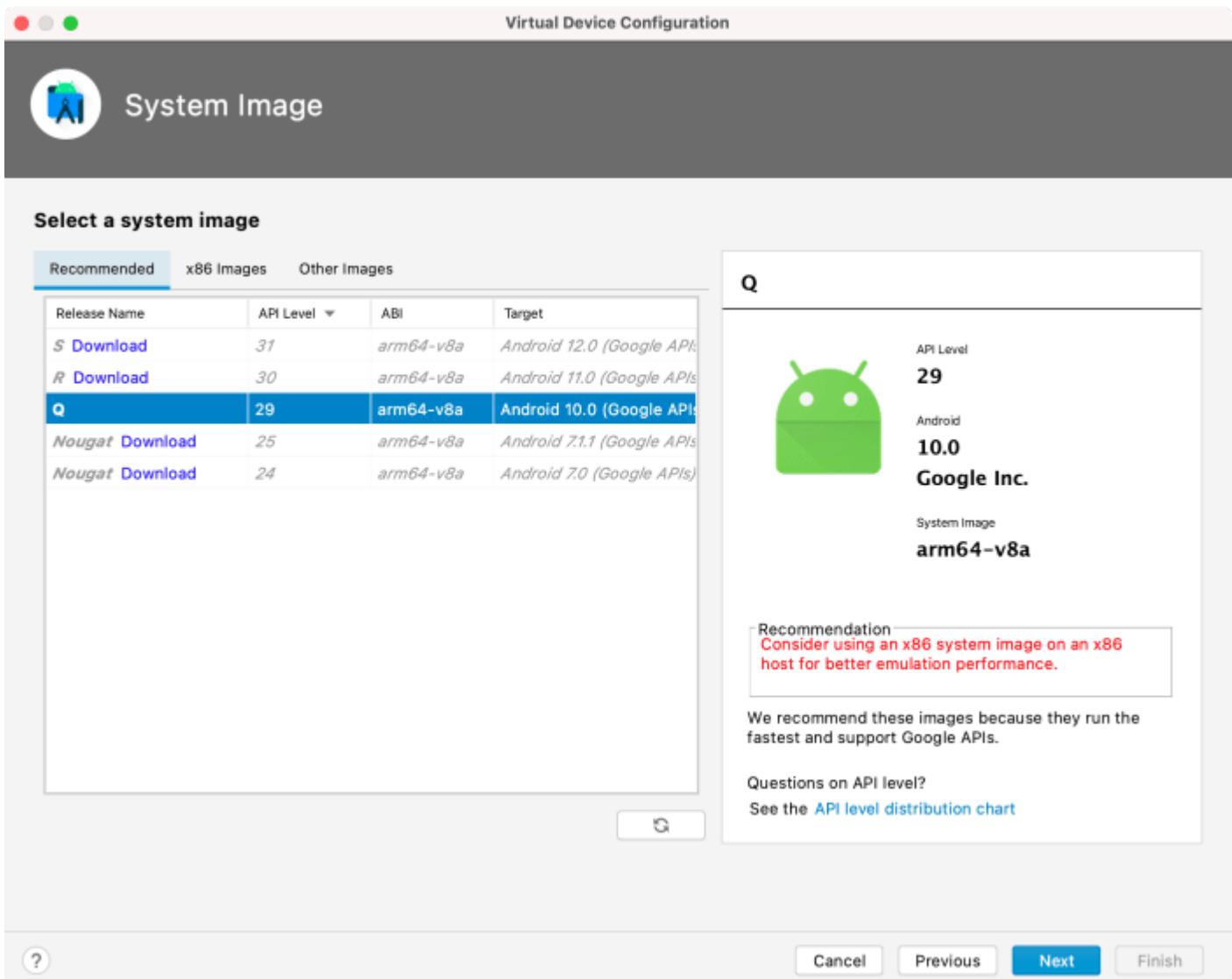
1080px

2340px

?

Cancel Previous Next Finish

Pick the target SDK version. This is the targetSdkVersion set in the [build.gradle](#) file.



There are some advanced settings we can toggle, but these are optional. Click finish.

Run the demo app

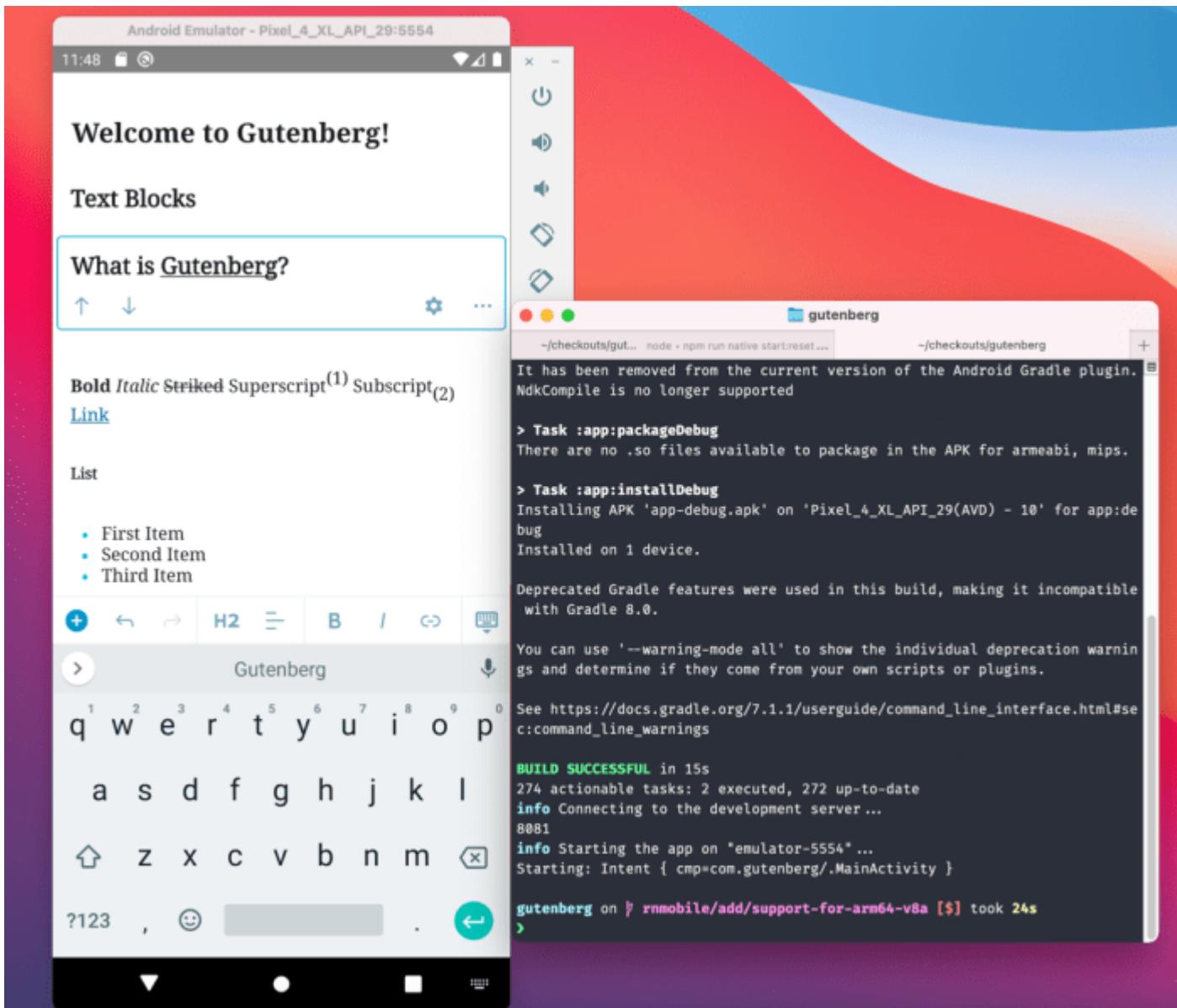
Start metro:

```
npm run native start:reset
```

In another terminal run the following to launch the demo app in the Android emulator (if the emulator isn't already running, it'll also be launched as part of this command):

```
npm run native android
```

After a bit of a wait, we'll see something like this:



Unit Tests

```
npm run test:native
```

Integration Tests

[Appium](#) has its own doctor tool. Run this with:

```
npx appium-doctor
```

```
● ● ● gutenberg
info AppiumDoctor Appium Doctor v.1.16.0
info AppiumDoctor ### Diagnostic for necessary dependencies starting ###
info AppiumDoctor ✓ The Node.js binary was found at: /Users/kerryliu/.volta/tools/image/node/14.17.6/bin/node
info AppiumDoctor ✓ Node version is 14.17.6
info AppiumDoctor ✓ Xcode is installed at: /Applications/Xcode.app/Contents/Developer
info AppiumDoctor ✓ Xcode Command Line Tools are installed in: /Applications/Xcode.app/Contents/Developer
info AppiumDoctor ✓ DevToolsSecurity is enabled.
info AppiumDoctor ✓ The Authorization DB is set up properly.
WARN AppiumDoctor ✘ Carthage was NOT found!
info AppiumDoctor ✓ HOME is set to: /Users/kerryliu
info AppiumDoctor ✓ ANDROID_HOME is set to: /Users/kerryliu/Library/Android/sdk
info AppiumDoctor ✓ JAVA_HOME is set to: /Applications/Android Studio.app/Contents/jre/Contents/Home
info AppiumDoctor Checking adb, android, emulator
info AppiumDoctor     'adb' is in /Users/kerryliu/Library/Android/sdk/platform-tools/adb
info AppiumDoctor     'android' is in /Users/kerryliu/Library/Android/sdk/tools/android
info AppiumDoctor     'emulator' is in /Users/kerryliu/Library/Android/sdk/emulator/emulator
info AppiumDoctor ✓ adb, android, emulator exist: /Users/kerryliu/Library/Android/sdk
info AppiumDoctor ✓ 'bin' subfolder exists under '/Applications/Android Studio.app/Contents/jre/Contents/Home'
info AppiumDoctor ### Diagnostic for necessary dependencies completed, one fix needed. ###
info AppiumDoctor
info AppiumDoctor ### Diagnostic for optional dependencies starting ###
WARN AppiumDoctor ✘ opencv4nodejs cannot be found.
WARN AppiumDoctor ✘ ffmpeg cannot be found
WARN AppiumDoctor ✘ mjpeg-consumer cannot be found.
WARN AppiumDoctor ✘ set-simulator-location is not installed
WARN AppiumDoctor ✘ idb and idb_companion are not installed
WARN AppiumDoctor ✘ applesimutils cannot be found
info AppiumDoctor ✓ ios-deploy is installed at: /Users/kerryliu/.volta/tools/image/node/14.17.6/bin/ios-deploy. Installed version is: 1.11.4
WARN AppiumDoctor ✘ bundletool.jar cannot be found
WARN AppiumDoctor ✘ gst-launch-1.0 and/or gst-inspect-1.0 cannot be found
info AppiumDoctor ### Diagnostic for optional dependencies completed, 8 fixes possible. ###
info AppiumDoctor
info AppiumDoctor ### Manual Fixes Needed ###
info AppiumDoctor The configuration cannot be automatically fixed, please do the following first:
WARN AppiumDoctor → [For lower than Appium 1.20.0] Please install Carthage. Visit https://github.com/Carthage/Carthage#installing-carthage for more information.
info AppiumDoctor
info AppiumDoctor ### Optional Manual Fixes ###
info AppiumDoctor The configuration can install optionally. Please do the following manually:
```

Resolve any required dependencies.

iOS Integration Tests

If we know we can run the iOS local environment without issue, E2Es for iOS are straightforward. Stop any running metro processes. This was launched previously with `npm run native start:reset`.

Then in terminal type:

```
npm run native test:e2e:ios:local
```

Passing a filename should also work to run a subset of tests:

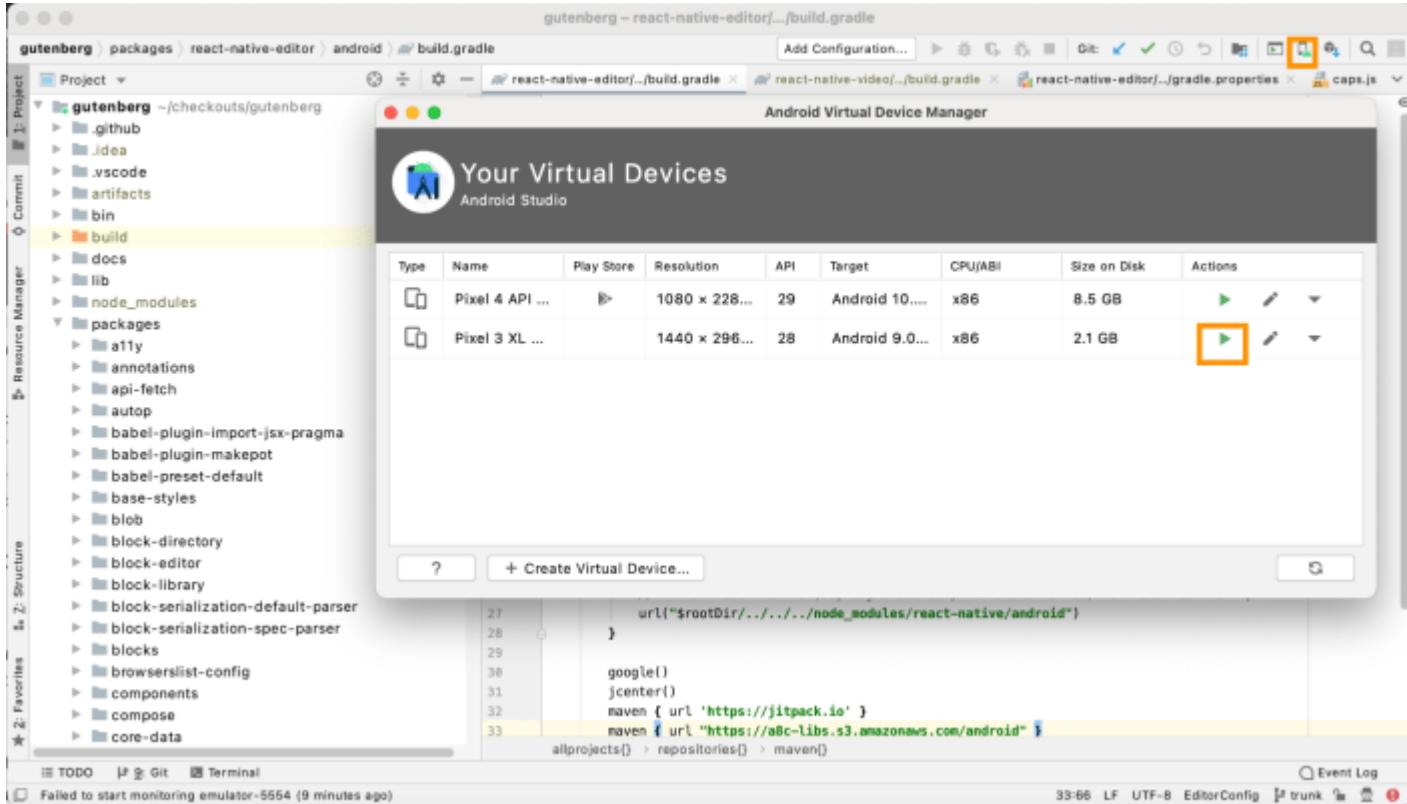
```
npm run native test:e2e:ios:local gutenberg-editor-paragraph.test.js
```

If all things go well, it should look like:

Android Integration Tests

Create a new virtual device() that matches the device specified in [packages/react-native-editor/device-tests/helpers/caps.js](#) At the time of this writing, this would be a Pixel 3 XL image, using Android 9 (API 28).

Start the virtual device first. Go back to the AVD by clicking on the phone icon, then click the green play button.



Make sure no metro processes are running. This was launched previously with `npm run native start:reset`.

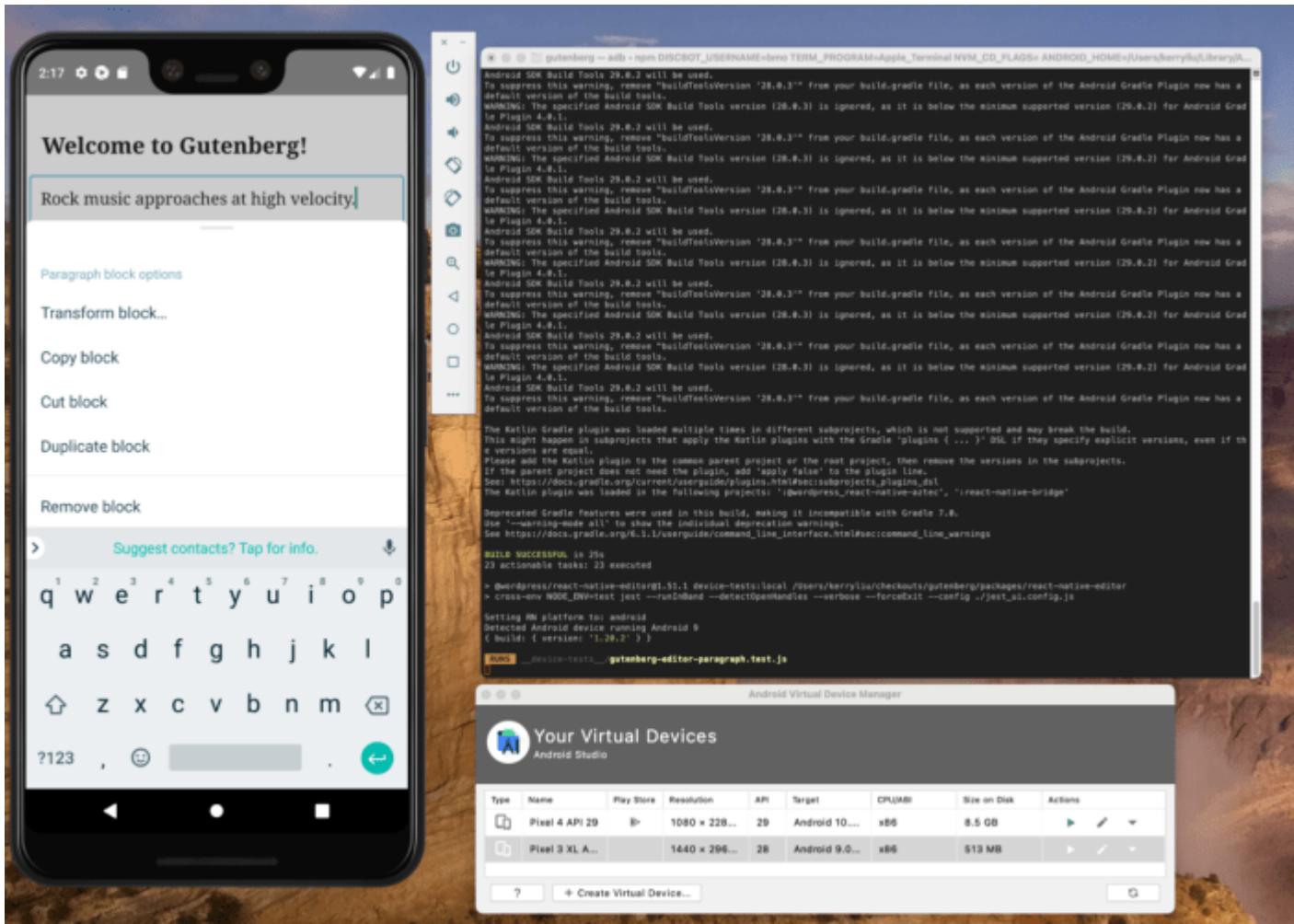
Then in a terminal run:

```
npm run native test:e2e:android:local
```

Passing a filename should also work to run a subset of tests:

```
npm run native test:e2e:android:local gutenberg-editor-paragraph.test.js
```

After a bit of a wait we should see:



First published

December 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Setup guide for React Native development \(macOS\)](#)

[Previous](#) [Getting Started for the React Native based Mobile Gutenberg](#) [Previous: Getting Started for the React Native based Mobile Gutenberg](#)

[Next](#) [React Native Integration Test Guide](#) [Next: React Native Integration Test Guide](#)

React Native Integration Test Guide

In this article

Table of Contents

- [What's an integration test?](#)

- [Anatomy of an integration test](#)
- [Setup](#)
- [Rendering](#)
 - [Using the Scoped Component approach](#)
 - [Using the Entire Editor approach](#)
- [Query elements](#)
 - [Use of find queries](#)
 - [within queries](#)
- [Fire events](#)
- [Expect correct element behaviour](#)
- [Cleanup](#)
- [Helpers](#)
- [Common flows](#)
 - [Query a block](#)
 - [Add a block](#)
 - [Open block settings](#)
 - [FlatList items](#)
 - [Sliders](#)
 - [Selecting an inner block](#)
- [Tools](#)
 - [Using the Accessibility Inspector](#)
- [Common pitfalls and caveats](#)
 - [False positives when omitting await before waitFor function](#)
 - [waitFor timeout](#)
 - [Replace current UI unit tests](#)
 - [Platform selection](#)

[↑ Back to top](#)

What's an integration test?

Integration testing is defined as a type of testing where different parts are tested as a group. In our case, the parts that we want to test are the different components that are required to be rendered for a specific block or editor logic. In the end, they are very similar to unit tests as they are run with the same command using the Jest library. The main difference is that for the integration tests, we're going to use a specific library [react-native-testing-library](#) for testing how the editor renders the different components.

Anatomy of an integration test

A test can be structured with the following parts:

- [Setup](#)
- [Rendering](#)
- [Query elements](#)
- [Fire events](#)
- [Expect correct element behaviour](#)
- [Cleanup](#)

We also include examples of common tasks as well as tips in the following sections:

- [Helpers](#)

- [Common flows](#)
- [Tools](#)
- [Common pitfalls and caveats](#)

Setup

This part usually is covered by using the Jest callbacks `beforeAll` and `beforeEach`, the purpose is to prepare everything that the test might require like registering blocks or mocking parts of the logic.

Here is an example of a common pattern if we expect all core blocks to be available:

```
beforeAll( () => {
  // Register all core blocks
  registerCoreBlocks();
} );
```

Rendering

Before introducing the testing logic, we have to render the components that we want to test on. Depending on if we want to use the scoped component or entire editor approach, this part will be different.

Using the Scoped Component approach

Here is an example of rendering the Cover block (extracted from [this code](#)):

```
// This import points to the index file of the block
import { metadata, settings, name } from '../index';

...

const setAttributes = jest.fn();
const attributes = {
  backgroundType: IMAGE_BACKGROUND_TYPE,
  focalPoint: { x: '0.25', y: '0.75' },
  hasParallax: false,
  overlayColor: { color: '#000000' },
  url: 'mock-url',
};

...

// Simplified tree to render Cover edit within slot
const CoverEdit = ( props ) => (
  <SlotFillProvider>
    <BlockEdit isSelected={ name } clientId={ 0 } { ...props } />
    <BottomSheetSettings isVisible />
  </SlotFillProvider>
);

const { getByText, findByText } = render(
```

```

<CoverEdit
    attributes={ {
        ...attributes,
        url: undefined,
        backgroundType: undefined,
    } }
    setAttributes={ setAttributes }
/>
);

```

Using the Entire Editor approach

Here is an example of rendering the Buttons block (extracted from [this code](#)):

```

const initialHtml = `<!-- wp:buttons -->
<div class="wp-block-buttons"><!-- wp:button {"style":{"border":{"radius":`+
<div class="wp-block-button"><a class="wp-block-button__link" style="border:<!-- /wp:button --></div>
<!-- /wp:buttons -->`;
const { getByLabelText } = initializeEditor( {
    initialHtml,
} );

```

Query elements

Once the components are rendered, it's time to query them. An important note about this topic is that we should test from the user's perspective, this means that ideally we should query by elements that the user has access to like texts or accessibility labels.

When querying we should follow this priority order:

1. `getByText`: querying by text is the closest flow we can do from the user's perspective, as text is the visual clue for them to identify elements.
2. `getByLabelText`: in some cases, we want to query elements that don't provide text so in this case we can fallback to the accessibility label.
3. `getById`: if none of the previous options fit and/or we don't have any visual element that we can rely upon, we have to fallback to a specific test id, which can be defined using the `testID` attribute (see [here](#) for an example).

Here are some examples:

```

const mediaLibraryButton = getByText( 'WordPress Media Library' );
const missingBlock = getByLabelText( /Unsupported Block\. Row 1/ );
const radiusSlider = getById( 'Slider Border Radius' );

```

Note that either a plain string or a regular expression can be passed into these queries. A regular expression is best for querying part of a string (e.g. any element whose accessibility label contains `Unsupported Block. Row 1`). Note that special characters such as `.` need to be escaped.

Use of find queries

After rendering the components or firing an event, side effects might happen due to potential state updates so the element we're looking for might not be yet rendered. In this case, we would need to wait for the element to be available and for this purpose, we can use the `find*` versions of query functions, which internally use `waitFor` and periodically check whether the element appeared or not.

Here are some examples:

```
const mediaLibraryButton = await findByText( 'WordPress Media Library' );
const missingBlock = await findByLabelText( /Unsupported Block\. Row 1/ );
const radiusSlider = await findByTestId( 'Slider Border Radius' );
```

In most cases we'll use the `find*` functions, but it's important to note that it should be restricted to those queries that actually require waiting for the element to be available.

within queries

It's also possible to query elements contained in other elements via the `within` function, here is an example:

```
const missingBlock = await findByLabelText( /Unsupported Block\. Row 1/ );
const translatedTableTitle = within( missingBlock ).getByText( 'Tabla' );
```

Fire events

As important as querying an element is to trigger events to simulate the user interaction, for this purpose we can use the `fireEvent` function ([documentation](#)).

Here is an example of a press event:

Press event:

```
fireEvent.press( settingsButton );
```

We can also trigger any type of event, including custom events, in the following example you can see how we trigger the `onValueChange` event ([code reference](#)) for the Slider component:

Custom event – `onValueChange`:

```
fireEvent( heightSlider, 'valueChange', '50' );
```

Expect correct element behaviour

After querying elements and firing events, we must verify that the logic works as expected. For this purpose we can use the same `expect` function from Jest that we use in unit tests. It is recommended to use the custom `toBeVisible` matcher to ensure the element is defined, is a valid React element, and visible.

Here is an example:

```
const translatedTableTitle = within( missingBlock ).getByText( 'Tabla' );
expect( translatedTableTitle ).toBeVisible();
```

Additionally when rendering the entire editor, we can also verify if the HTML output is what we expect:

```
expect( getEditorHtml() ).toBe(
    '<!-- wp:spacer {"height":50} -->\n<div style="height:50px" aria-hidden="true">\n</div>');
```

Cleanup

And finally, we have to clean up any potential modifications we've made that could affect the following tests. Here is an example of a typical cleanup after registering blocks that implies unregistering all blocks:

```
afterAll( () => {
    // Clean up registered blocks
    getBlockTypes().forEach( ( block ) => {
        unregisterBlockType( block.name );
    } );
});
```

Helpers

In the spirit of making easier writing integration tests for the native version, you can find a list of helper functions in [this README](#).

Common flows

Query a block

A common way to query a block is by its accessibility label, here is an example:

```
const spacerBlock = await waitFor( () =>
    getByLabelText( /Spacer Block/. Row 1/ )
);
```

For further information about the accessibility label of a block, you can check the code of the [function `getAccessibleBlockLabel`](#).

Add a block

Here is an example of how to insert a Paragraph block:

```
// Open the inserter menu
fireEvent.press( await findByLabelText( 'Add block' ) );

const blockList = getByTestId( 'InserterUI-Blocks' );
// onScroll event used to force the FlatList to render all items
fireEvent.scroll( blockList, {
    nativeEvent: {
```

```

        contentOffset: { y: 0, x: 0 },
        contentSize: { width: 100, height: 100 },
        layoutMeasurement: { width: 100, height: 100 },
    },
} );

```

```

// Insert a Paragraph block
fireEvent.press( await findByText( `Paragraph` ) );

```

[Open block settings](#)

The block settings can be accessed by tapping the “Open Settings” button after selecting the block, here is an example:

```

fireEvent.press( block );

const settingsButton = await findByLabelText( 'Open Settings' );
fireEvent.press( settingsButton );

```

Using the Scoped Component approach

When using the scoped component approach, we need first to render the `SlotFillProvider` and the `BottomSheetSettings` (note that we’re passing the `isVisible` prop to force the bottom sheet to be displayed) along with the block:

```

<SlotFillProvider>
    <BlockEdit isSelected name={ name } clientId={ 0 } { ...props } />
    <BottomSheetSettings isVisible />
</SlotFillProvider>

```

See examples:

- [Cover block](#)

[FlatList items](#)

The `FlatList` component renders its items depending on the scroll position, the view and content size. This means that when rendering this component it might happen that some of the items can’t be queried because they haven’t been rendered yet. To address this issue we have to explicitly fire an event to make the `FlatList` render all the items.

Here is an example of the `FlatList` used for rendering the block list in the inserter menu:

```

const blockList = getByTestId( 'InserterUI-Blocks' );
// onScroll event used to force the FlatList to render all items
fireEvent.scroll( blockList, {
    nativeEvent: {
        contentOffset: { y: 0, x: 0 },
        contentSize: { width: 100, height: 100 },
        layoutMeasurement: { width: 100, height: 100 },
    },
} );

```

Sliders

Sliders found in bottom sheets should be queried using their testID:

```
const radiusSlider = await findByTestId('Slider Border Radius');
fireEvent(radiusSlider, 'valueChange', '30');
```

Note that a slider's testID is "Slider" + label. So for a slider with a label of "Border Radius", testID is "Slider Border Radius".

Selecting an inner block

One caveat when adding blocks is that if they contain inner blocks, these inner blocks are not rendered. The following example shows how we can make a Buttons block render its inner Button blocks (assumes we've already obtained a reference to the Buttons block as buttonsBlock):

```
const innerBlockListWrapper = await within(buttonsBlock).findByTestId(
  'block-list-wrapper'
);
fireEvent(innerBlockListWrapper, 'layout', {
  nativeEvent: {
    layout: {
      width: 100,
    },
  },
} );
const buttonInnerBlock = await within(buttonsBlock).findByLabelText(
  '/Button Block\\ Row 1'
);
fireEvent.press(buttonInnerBlock);
```

Tools

Using the Accessibility Inspector

If you have trouble locating an element's identifier, you may wish to use Xcode's Accessibility Inspector. Most identifiers are cross-platform, so even though the tests are run on Android by default, the Accessibility Inspector can be used to find the right identifier.

Accessibility Inspector

Simulator - iPhone 12 (iOS 14.5) **Change this to match sim**

Inspected Element:
Open Settings, Button

Navigation

Basic

Label: Open Settings  **This is the label you can query using getByText**

Value: None

Traits: Button

Identifier: None

Actions

Activate: **Perform**

Element

Class: RCTView

Address: 0x7f8cb3043ee0

Controller: None

Hierarchy

- ✓ Gutenberg
 - ✓ <UIWindow>
 - ✓ <UITransitionView>
 - ✓ <UIDropShadowView>
 - ✓ <UILayoutContainerView>
 - ✓ <UINavigationTransitionView>
 - ✓ <UIViewControllerWrapperView>
 - ✓ <RCTRootView>

Common pitfalls and caveats

False positives when omitting await before waitFor function

Omitting the `await` before a `waitFor` can lead to scenarios where tests pass but are not testing the intended behaviour. For example, if `toBeDefined` is used to assert the result of a call to `waitFor`, the assertion will pass because `waitFor` returns a value, even though it is not the `ReactTestInstance` we meant to check for. For this reason, it is recommended to use the custom matcher `toBeVisible` which guards against this type of false positive.

waitFor timeout

The default timeout for the `waitFor` function is set to 1000 ms, so far this value is enough for all the render logic we're testing, however, if while testing we notice that an element requires more time to be rendered we should increase it.

Replace current UI unit tests

Some components already have unit tests that cover component rendering, although it's not mandatory, in these cases, it would be nice to analyze the potential migration to an integration test.

In case we want to keep both, we'll add the word "integration" to the integration test file to avoid naming conflicts, here is an example: [packages/block-library/src/missing/test/edit-integration.native.js](#).

Platform selection

By default, all tests run in Jest use the Android platform, so in case we need to test a specific behaviour related to a different platform, we would need to support platform test files.

In case we only need to test logic controlled by the `Platform` object, we can mock the module with the following code (in this case it's changing the platform to iOS):

```
jest.mock( 'Platform', () => {
  const Platform = jest.requireActual( 'Platform' );
  Platform.OS = 'ios';
  Platform.select = jest.fn().mockImplementation( ( select ) => {
    const value = select[ Platform.OS ];
    return ! value ? select.default : value;
  } );
  return Platform;
} );
```

First published

December 7, 2021

Last updated

January 29, 2024

Edit article

React Native Internationalization Guide

In this article

[Table of Contents](#)

- [Extract strings only used in the native platform](#)
 - [NPM command](#)
- [Providing own translations \(for strings only used in native platform\)](#)
- [Fetch translations \(for strings used in web and native platforms\)](#)
 - [NPM command](#)

[↑ Back to top](#)

The native version of the editor references two types of string:

1. Strings used in web and native platforms.
2. Strings used only in the native platform.

Regarding the first type, these strings are translated following the same process described for the web version in [this guide](#), however for the latter, it's required to provide your own translations.

[Extract strings only used in the native platform](#)

In order to identify these strings, you can use the script `extract-used-strings` located in `packages/react-native-editor/bin/extract-used-strings.js` to generate a JSON object that contains all the strings referenced including the platforms where they are used, as well as the files that reference it. Here you can see the format:

```
{  
  "gutenberg": {  
    "<string>": {  
      "string": String value.  
      "stringPlural": String value with its plural form. [optional]  
      "comments": Comments for translators. [default value is an empty string]  
      "reference": Array containing the paths of the source files that reference this string.  
      "platforms": Array containing the platforms where the string is used.  
    },  
    ...  
  },  
  "other-domain-plugin": {  
    ...  
  },  
  ...  
}
```

This command also supports passing extra plugins, in case the React Native bundle of the editor is generated including other plugins.

It's important to note that the JSON object contains all used strings, so in order to identify the ones only used in the native platform, it's required to provide your own script/process to extract them. This can easily be done by going through the strings and filtering out the ones that include the "web" platform.

NPM command

Extract used strings:

```
npm run native i18n:extract-used-strings -- "$PWD/used-strings.json"
```

NOTE: We need to pass absolute paths, otherwise it uses packages/react-native-editor as root path for relative paths.

Extract used strings including extra plugins:

```
npm run native i18n:extract-used-strings -- "$PWD/used-strings.json" "doma
```

Providing own translations (for strings only used in native platform)

Once you have the list of used strings in the native platform, the strings have to be translated, however, this process is out of the scope of the native version so you have to provide your own translations.

The process for injecting the translations data into the editor is via the `translations` initial prop which is passed to the editor during its initialization:

- [Android reference](#)
- [iOS reference](#)

The mechanism for integrating the provided translations to the mobile client, via the mentioned `translations` initial prop, is not described here, as it's specific to the mobile client and could be achieved in different ways. Nevertheless, it's important that they're provided by the mentioned initial prop, as the editor is in charge of merging them with the translations already included in the editor.

NOTE: Keep in mind that those strings that match with ones already included in the editor will be overridden.

Fetch translations (for strings used in web and native platforms)

A translation file is basically a JSON object that contains key-value items with the translation for each individual string. This content is fetched from [translate.wordpress.org](#) that holds translations for WordPress and a list of different plugins like Gutenberg.

These files can be cached under a folder and optimized. Additionally, an index file is generated that acts as the entry point to import and fetches the plugin translations.

Fetched translations contain all the translatable plugin strings, including those not used in the native version of the editor. The file sizes, however, can be reduced by filtering out the strings not referenced in the used strings JSON file.

By default, when installing dependencies, un-optimized translations might be downloaded for Gutenberg and located in the `i18n-cache` folder if the cache is not present.

The strings included in these translation files are imported in the editor upon its initialization ([reference](#)) and will be merged with the extra translations provided by the `translations` initial prop.

[NPM command](#)

Fetch un-optimized translations:

```
npm run native i18n:fetch-translations -- "gutenberg" <OUTPUT_PATH>
```

NOTE: We need to pass absolute paths, otherwise it uses packages/react-native-editor as root path for relative paths.

Fetch optimized translations:

```
npm run native i18n:fetch-translations -- "gutenberg" <OUTPUT_PATH> <USED_
```

First published

January 18, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: React Native Internationalization Guide](#)

[Previous React Native Integration Test Guide](#) [Previous: React Native Integration Test Guide](#)
[Next Backward Compatibility](#) [Next: Backward Compatibility](#)

Backward Compatibility

In this article

[Table of Contents](#)

- [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](#)

[↑ Back to top](#)

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) is 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.

First published

February 5, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Backward Compatibility”](#)

[Previous React Native Internationalization Guide](#) [Previous: React Native Internationalization Guide](#)

[Next Deprecations](#) [Next: Deprecations](#)

Deprecations

In this article

Table of Contents

- [Unreleased](#)
- [11.0.0](#)
- [10.3.0](#)
- [9.7.0](#)
- [8.6.0](#)
- [8.3.0](#)
- [5.5.0](#)
- [5.4.0](#)
- [5.3.0](#)
- [5.2.0](#)
- [4.5.0](#)
- [4.4.0](#)
- [4.3.0](#)
- [4.2.0](#)
- [4.1.0](#)
- [4.0.0](#)
- [3.9.0](#)
- [3.8.0](#)
- [3.7.0](#)
- [3.6.0](#)
- [3.5.0](#)
- [3.4.0](#)
- [3.3.0](#)
- [3.2.0](#)
- [3.1.0](#)
- [3.0.0](#)
- [2.8.0](#)

- [2.7.0](#)
- [2.6.0](#)
- [2.5.0](#)
- [2.4.0](#)

[↑ Back to top](#)

For features included in the Gutenberg plugin, the deprecation policy is intended to support backward compatibility for two minor plugin releases, when possible. Features and code included in a stable release of WordPress are not included in this deprecation timeline, and are instead subject to the [versioning policies of the WordPress project](#). The current deprecations are listed below and are grouped by *the version at which they will be removed completely*. If your plugin depends on these behaviors, you must update to the recommended alternative before the noted version.

Unreleased

- `wp.blocks.isValidBlockContent` has been removed. Please use `wp.blocks.validateBlock` instead.

11.0.0

- `wp.blocks.registerBlockTypeFromMetadata` method has been removed. Use `wp.blocks.registerBlockType` method instead.

10.3.0

- Passing a tuple of components with `as` prop to `ActionItem.Slot` component is no longer supported. Please pass a component with `as` prop instead. Example:

```
“diff
<ActionItem.Slot
name=”my/slot”
label={ __( ‘My slot’ ) }
  ° as={ [ MenuGroup, MenuItem ] }
  ° as={ MenuGroup }
/>
“
```

9.7.0

- `leftSidebar` prop in `InterfaceSkeleton` component has been removed. Use `secondarySidebar` prop instead.

8.6.0

- Block API integration with [Block Context](#) was updated. When registering a block use `useContext` and `providesContext` pair in JavaScript files and `uses_context` and `provides_context` pair in PHP files instead of previous pair `context` and `providesContext`.

8.3.0

- The PHP function `gutenberg_get_post_from_context` has been removed. Use [Block Context](#) instead.
- The old Block Pattern APIs `register_pattern/unregister_pattern` have been removed. Use the [new functions](#) instead.

5.5.0

- The PHP function `gutenberg_init` has been removed.
- The PHP function `is_gutenberg_page` has been removed. Use [WP_Screen::is_block_editor](#) instead.
- The PHP function `the_gutenberg_project` has been removed.
- The PHP function `gutenberg_default_post_format_template` has been removed.
- The PHP function `gutenberg_get_available_image_sizes` has been removed.
- The PHP function `gutenberg_get_autosave_newer_than_post_save` has been removed.
- The PHP function `gutenberg_editor_scripts_and_styles` has been removed.

5.4.0

- The PHP function `gutenberg_load_plugin_textdomain` has been removed.
- The PHP function `gutenberg_get_jed_locale_data` has been removed.
- The PHP function `gutenberg_load_locale_data` has been removed.

5.3.0

- The PHP function
`gutenberg_redirect_to_classic_editor_when_saving_posts` has been removed.
- The PHP function `gutenberg_revisions_link_to_editor` has been removed.
- The PHP function
`gutenberg_remember_classic_editor_when_saving_posts` has been removed.
- The PHP function `gutenberg_can_edit_post_type` has been removed. Use [use_block_editor_for_post_type](#) instead.
- The PHP function `gutenberg_can_edit_post` has been removed. Use [use_block_editor_for_post](#) instead.

5.2.0

- The PHP function `gutenberg_parse_blocks` has been removed. Use [parse_blocks](#) instead.
- The PHP function `get_dynamic_blocks_regex` has been removed.
- The PHP function `gutenberg_render_block` has been removed. Use [render_block](#) instead.
- The PHP function `strip_dynamic_blocks` has been removed. For use in excerpt preparation, consider [excerpt_remove_blocks](#) instead.
- The PHP function `strip_dynamic_blocks_add_filter` has been removed.

- The PHP function `strip_dynamic_blocks_remove_filter` has been removed.
- The PHP function `gutenberg_post_has_blocks` has been removed. Use [`has_blocks`](#) instead.
- The PHP function `gutenberg_content_has_blocks` has been removed. Use [`has_blocks`](#) instead.
- The PHP function `gutenberg_register_rest_routes` has been removed.
- The PHP function `gutenberg_add_taxonomy_visibility_field` has been removed.
- The PHP function `gutenberg_get_taxonomy_visibility_data` has been removed.
- The PHP function `gutenberg_add_permalink_template_to_posts` has been removed.
- The PHP function `gutenberg_add_block_format_to_post_content` has been removed.
- The PHP function `gutenberg_add_target_schema_to_links` has been removed.
- The PHP function `gutenberg_register_post_prepare_functions` has been removed.
- The PHP function `gutenberg_silence_rest_errors` has been removed.
- The PHP function `gutenberg_filter_post_type_labels` has been removed.
- The PHP function `gutenberg_preload_api_request` has been removed. Use [`rest_preload_api_request`](#) instead.
- The PHP function `gutenberg_remove_wpcom_markdown_support` has been removed.
- The PHP function `gutenberg_add_gutenberg_post_state` has been removed.
- The PHP function `gutenberg_bulk_post_updated_messages` has been removed.
- The PHP function `gutenberg_kses_allowedtags` has been removed.
- The PHP function `gutenberg_add_responsive_body_class` has been removed.
- The PHP function `gutenberg_add_edit_link_filters` has been removed.
- The PHP function `gutenberg_add_edit_link` has been removed.
- The PHP function `gutenberg_block_bulk_actions` has been removed.
- The PHP function `gutenberg_replace_default_add_new_button` has been removed.
- The PHP function `gutenberg_content_block_version` has been removed. Use [`block_version`](#) instead.
- The PHP function `gutenberg_get_block_categories` has been removed. Use [`get_block_categories`](#) instead.
- The PHP function `register_tinymce_scripts` has been removed. Use [`wp_register_tinymce_scripts`](#) instead.
- The PHP function `gutenberg_register_post_types` has been removed.
- The `gutenberg` theme support option has been removed. Use [`align-wide`](#) instead.
- The PHP function `gutenberg_prepare_blocks_for_js` has been removed. Use [`get_block_editor_server_block_settings`](#) instead.
- The PHP function `gutenberg_load_list_reusable_blocks` has been removed.
- The PHP function `_gutenberg_utf8_split` has been removed. Use `_mb_substr` instead.
- The PHP function `gutenberg_disable_editor_settings_wpautop` has been removed.
- The PHP function `gutenberg_add_rest_nonce_to_heartbeat_response_headers` has been removed.

- The PHP function `gutenberg_check_if_classic_needs_warning_about_blocks` has been removed.
- The PHP function `gutenberg_warn_classic_about_blocks` has been removed.
- The PHP function `gutenberg_show_privacy_policy_help_text` has been removed.
- The PHP function `gutenberg_common_scripts_and_styles` has been removed. Use [`wp_common_block_scripts_and_styles`](#) instead.
- The PHP function `gutenberg_enqueue_registered_block_scripts_and_styles` has been removed. Use [`wp_enqueue_registered_block_scripts_and_styles`](#) instead.
- The PHP function `gutenberg_meta_box_save` has been removed.
- The PHP function `gutenberg_meta_box_save_redirect` has been removed.
- The PHP function `gutenberg_filter_meta_boxes` has been removed.
- The PHP function `gutenberg_intercept_meta_box_render` has been removed.
- The PHP function `gutenberg_override_meta_box_callback` has been removed.
- The PHP function `gutenberg_show_meta_box_warning` has been removed.
- The PHP function `the_gutenberg_metaboxes` has been removed. Use [`the_block_editor_meta_boxes`](#) instead.
- The PHP function `gutenberg_meta_box_post_form_hidden_fields` has been removed. Use [`the_block_editor_meta_box_post_form_hidden_fields`](#) instead.
- The PHP function `gutenberg_toggle_custom_fields` has been removed.
- The PHP function `gutenberg_collect_meta_box_data` has been removed. Use [`register_and_do_post_meta_boxes`](#) instead.
- `window._wpLoadGutenbergEditor` has been removed. Use `window._wpLoadBlockEditor` instead. Note: This is a private API, not intended for public use. It may be removed in the future.
- The PHP function `gutenberg_get_script_polyfill` has been removed. Use [`wp_get_script_polyfill`](#) instead.
- The PHP function `gutenberg_add_admin_body_class` has been removed. Use the `.block-editor-page` class selector in your stylesheets if you need to scope styles to the block editor screen.

4.5.0

- `Dropdown.refresh()` has been deprecated as the contained `Popover` is now automatically refreshed.
- `wp.editor.PostPublishPanelToggle` has been deprecated in favor of `wp.editor.PostPublishButton`.

4.4.0

- `wp.date.getSettings` has been removed. Please use `wp.date.__experimentalGetSettings` instead.
- `wp.compose.remountOnPropChange` has been removed.
- The following editor store actions have been removed: `createNotice`, `removeNotice`, `createSuccessNotice`, `createInfoNotice`, `createErrorNotice`, `createWarningNotice`. Use the equivalent actions by the same name from the `@wordpress/notices` module.

- The `id` prop of `wp.nux.DotTip` has been removed. Please use the `tipId` prop instead.
- `wp.blocks.isValidBlock` has been removed. Please use `wp.blocks.isValidBlockContent` instead but keep in mind that the order of params has changed.
- `wp.data registry.registerReducer` has been deprecated. Use `registry.registerStore` instead.
- `wp.data registry.registerSelectors` has been deprecated. Use `registry.registerStore` instead.
- `wp.data registry.registerActions` has been deprecated. Use `registry.registerStore` instead.
- `wp.data registry.registerResolvers` has been deprecated. Use `registry.registerStore` instead.
- `moment` has been removed from the public API for the date module.

4.3.0

- `isEditorSidebarPanelOpened` selector (`core/edit-post`) has been removed. Please use `isEditorPanelEnabled` instead.
- `toggleGeneralSidebarEditorPanel` action (`core/edit-post`) has been removed. Please use `toggleEditorPanelOpened` instead.
- `wp.components.PanelColor` component has been removed. Please use `wp.editor.PanelColorSettings` instead.
- `wp.editor.PanelColor` component has been removed. Please use `wp.editor.PanelColorSettings` instead.

4.2.0

- Writing resolvers as async generators has been removed. Use the controls plugin instead.
- `wp.components.AccessibleSVG` component has been removed. Please use `wp.components.SVG` instead.
- The `wp.editor.UnsavedChangesWarning` component no longer accepts a `forceIsDirty` prop.
- `setActiveMetaBoxLocations` action (`core/edit-post`) has been removed.
- `initializeMetaBoxState` action (`core/edit-post`) has been removed.
- `wp.editPost.initializeEditor` no longer returns an object. Use the `setActiveMetaBoxLocations` action (`core/edit-post`) in place of the existing object's `initializeMetaBoxes` function.
- `setMetaBoxSavedData` action (`core/edit-post`) has been removed.
- `getMetaBoxes` selector (`core/edit-post`) has been removed. Use `getActiveMetaBoxLocations` selector (`core/edit-post`) instead.
- `getMetaBox` selector (`core/edit-post`) has been removed. Use `isMetaBoxLocationActive` selector (`core/edit-post`) instead.
- Attribute type coercion has been removed. Omit the source to preserve type via serialized comment demarcation.
- `mediaDetails` in object passed to `onChange` callback of `wp.editor.mediaUpload`. Please use `media_details` property instead.
- `wp.components.CodeEditor` has been removed. Used `wp.codeEditor` directly instead.
- `wp.blocks.setUnknownTypeHandlerName` has been removed. Please use `setFreeformContentHandlerName` and `setUnregisteredTypeHandlerName` instead.

- `wp.blocks.getUnknownTypeHandlerName` has been removed. Please use `getFreeformContentHandlerName` and `getUnregisteredTypeHandlerName` instead.
- The Reusable blocks Data API was marked as experimental as it's subject to change in the future.

4.1.0

- `wp.data.dispatch('core/editor').checkTemplateValidity` has been removed. Validity is verified automatically upon block reset.

4.0.0

- `wp.editor.RichTextProvider` has been removed. Please use `wp.data.select('core/editor')` methods instead.
- `wp.components.Draggable` as a DOM node drag handler has been removed. Please, use `wp.components.Draggable` as a wrap component for your DOM node drag handler.
- `wp.i18n.getI18n` has been removed. Use `__`, `_x`, `_n`, or `_nx` instead.
- `wp.i18n.dcngettext` has been removed. Use `__`, `_x`, `_n`, or `_nx` instead.

3.9.0

- `RichText.getSettings` prop has been removed. The `unstableGetSettings` prop is available if continued use is required. Unstable APIs are strongly discouraged to be used, and are subject to removal without notice.
- `RichText.onSetup` prop has been removed. The `unstableOnSetup` prop is available if continued use is required. Unstable APIs are strongly discouraged to be used, and are subject to removal without notice.
- `wp.editor.getColorName` has been removed. Please use `wp.editor.getColorObjectByColorValue` instead.
- `wp.editor.getColorClass` has been renamed. Please use `wp.editor.getColorClassName` instead.
- `value` property in color objects passed by `wp.editor.withColors` has been removed. Please use `color` property instead.
- The Subheading block has been removed. Please use the Paragraph block instead.
- `wp.blocks.getDefaultBlockForPostFormat` has been removed.

3.8.0

- `wp.components.withContext` has been removed. Please use `wp.element.createContext` instead. See: <https://reactjs.org/docs/context.html>.
- `wp.coreBlocks.registerCoreBlocks` has been removed. Please use `wp.blockLibrary.registerCoreBlocks` instead.
- `wp.editor.DocumentTitle` component has been removed.
- `getDocumentTitle` selector (`core/editor`) has been removed.

3.7.0

- `wp.components.withAPIData` has been removed. Please use the Core Data module or `wp.apiFetch` directly instead.
- `wp.data.dispatch("core").receiveTerms` has been deprecated. Please use `wp.data.dispatch("core").receiveEntityRecords` instead.
- `getCategories` resolver has been deprecated. Please use `getEntityRecords` resolver instead.
- `wp.data.select("core").getTerms` has been deprecated. Please use `wp.data.select("core").getEntityRecords` instead.
- `wp.data.select("core").getCategories` has been deprecated. Please use `wp.data.select("core").getEntityRecords` instead.
- `wp.data.select("core").isRequestingCategories` has been deprecated. Please use `wp.data.select("core/data").isResolving` instead.
- `wp.data.select("core").isRequestingTerms` has been deprecated. Please use `wp.data.select("core").isResolving` instead.
- `wp.data.restrictPersistence`, `wp.data.setPersistenceStorage` and `wp.data.setupPersistence` has been removed. Please use the data persistence plugin instead.

3.6.0

- `wp.editor.editorMediaUpload` has been removed. Please use `wp.editor.mediaUpload` instead.
- `wp.utils.getMimeTypesArray` has been removed.
- `wp.utils.mediaUpload` has been removed. Please use `wp.editor.mediaUpload` instead.
- `wp.utils.preloadImage` has been removed.
- `supports.wideAlign` has been removed from the Block API. Please use `supports.alignWide` instead.
- `wp.blocks.isSharedBlock` has been removed. Use `wp.blocks.isReusableBlock` instead.
- `fetchSharedBlocks` action (`core/editor`) has been removed. Use `fetchReusableBlocks` instead.
- `receiveSharedBlocks` action (`core/editor`) has been removed. Use `receiveReusableBlocks` instead.
- `saveSharedBlock` action (`core/editor`) has been removed. Use `saveReusableBlock` instead.
- `deleteSharedBlock` action (`core/editor`) has been removed. Use `deleteReusableBlock` instead.
- `updateSharedBlockTitle` action (`core/editor`) has been removed. Use `updateReusableBlockTitle` instead.
- `convertBlockToSaved` action (`core/editor`) has been removed. Use `convertBlockToReusable` instead.
- `getSharedBlock` selector (`core/editor`) has been removed. Use `getReusableBlock` instead.
- `isSavingSharedBlock` selector (`core/editor`) has been removed. Use `isSavingReusableBlock` instead.
- `isFetchingSharedBlock` selector (`core/editor`) has been removed. Use `isFetchingReusableBlock` instead.
- `getSharedBlocks` selector (`core/editor`) has been removed. Use `getReusableBlocks` instead.

3.5.0

- `wp.components.ifCondition` has been removed. Please use `wp.compose.ifCondition` instead.
- `wp.components.withGlobalEvents` has been removed. Please use `wp.compose.withGlobalEvents` instead.
- `wp.components.withInstanceId` has been removed. Please use `wp.compose.withInstanceId` instead.
- `wp.components.withSafeTimeout` has been removed. Please use `wp.compose.withSafeTimeout` instead.
- `wp.components.withState` has been removed. Please use `wp.compose.withState` instead.
- `wp.element.pure` has been removed. Please use `wp.compose.pure` instead.
- `wp.element.compose` has been removed. Please use `wp.compose.compose` instead.
- `wp.element.createHigherOrderComponent` has been removed. Please use `wp.compose.createHigherOrderComponent` instead.
- `wp.utils.buildTermsTree` has been removed.
- `wp.utils.decodeEntities` has been removed. Please use `wp.htmlEntities.decodeEntities` instead.
- All references to a block's `uid` have been replaced with equivalent props and selectors for `clientId`.
- The `wp.editor.MediaPlaceholder` component `onSelectUrl` prop has been renamed to `onSelectURL`.
- The `wp.editor.UrlInput` component has been renamed to `wp.editor.URLInput`.
- The Text Columns block has been removed. Please use the Columns block instead.
- InnerBlocks grouped layout is removed. Use intermediary nested inner blocks instead. See Columns / Column block for reference implementation.
- RichText explicit `element` format removed. Please use the compatible `children` format instead.

3.4.0

- `focusOnMount` prop in the Popover component has been changed from Boolean-only to an enum-style property that accepts `"firstElement"`, `"container"`, or `false`. Please convert any `<Popover focusOnMount />` usage to `<Popover focusOnMount="firstElement" />`.
- `wp.utils.keycodes` utilities are removed. Please use `wp.keycodes` instead.
- Block id prop in `edit` function removed. Please use block `clientId` prop instead.
- `property` source removed. Please use equivalent `text`, `html`, or `attribute` source, or `comment` attribute instead.

3.3.0

- `useOnce: true` has been removed from the Block API. Please use `supports.multiple: false` instead.
- Serializing components using `componentWillMount` lifecycle method. Please use the constructor instead.
- `blocks.Autocomplete.completers` filter removed. Please use `editor.Autocomplete.completers` instead.
- `blocks.BlockEdit` filter removed. Please use `editor.BlockEdit` instead.

- `blocks.BlockListBlock` filter removed. Please use `editor.BlockListBlock` instead.
- `blocks.MediaUpload` filter removed. Please use `editor.MediaUpload` instead.

3.2.0

- `wp.data.withRehydration` has been renamed to `wp.data.withRehydration`.
- The `wp.editor.ImagePlaceholder` component is removed. Please use `wp.editor.MediaPlaceholder` instead.
- `wp.utils.deprecated` function removed. Please use `wp.deprecated` instead.
- `wp.utils.blob` removed. Please use `wp.blob` instead.
- `getInserterItems`: the `allowedBlockTypes` argument was removed and the `parentUID` argument was added.
- `getRecentInserterItems` selector removed. Please use `getInserterItems` instead.
- `getSupportedBlocks` selector removed. Please use `canInsertBlockType` instead.

3.1.0

- All components in `wp.blocks.*` are removed. Please use `wp.editor.*` instead.
- `wp.blocks.withEditorSettings` is removed. Please use the data module to access the editor settings `wp.data.select("core/editor").getEditorSettings()`.
- All DOM utils in `wp.utils.*` are removed. Please use `wp.dom.*` instead.
- `isPrivate: true` has been removed from the Block API. Please use `supports.inserter: false` instead.
- `wp.utils.isExtraSmall` function removed. Please use `wp.viewport` module instead.
- `getEditedPostExcerpt` selector removed (`core/editor`). Use `getEditedPostAttribute('excerpt')` instead.

3.0.0

- `wp.blocks.registerCoreBlocks` function removed. Please use `wp.coreBlocks.registerCoreBlocks` instead.
- Raw TinyMCE event handlers for `RichText` have been deprecated. Please use [documented props](#), ancestor event handler, or `onSetup` access to the internal editor instance event hub instead.

2.8.0

- Original autocomplete interface in `wp.components.AutoComplete` updated. Please use latest autocomplete interface instead. See [autocomplete](#) for more info.
- `getInserterItems`: the `allowedBlockTypes` argument is now mandatory.
- `getRecentInserterItems`: the `allowedBlockTypes` argument is now mandatory.

2.7.0

- `wp.element.getWrapperDisplayName` function removed. Please use `wp.element.createHigherOrderComponent` instead.

2.6.0

- `wp.blocks.getBlockDefaultClassname` function removed. Please use `wp.blocks.getBlockDefaultClassName` instead.
- `wp.blocks.Editable` component removed. Please use the `wp.blocks.RichText` component instead.

2.5.0

- Returning raw HTML from block `save` is unsupported. Please use the `wp.element.RawHTML` component instead.
- `wp.data.query` higher-order component removed. Please use `wp.data.withSelect` instead.

2.4.0

- `wp.blocks.BlockDescription` component removed. Please use the `description` block property instead.
- `wp.blocks.InspectorControls.*` components removed. Please use `wp.components.*` components instead.
- `wp.blocks.source.*` matchers removed. Please use the declarative attributes instead. See [block attributes](#) for more info.
- `wp.data.select('selector', ...args)` removed. Please use `wp.data.select(reducerKey').*` instead.
- `wp.blocks.MediaUploadButton` component removed. Please use `wp.blocks.MediaUpload` component instead.

First published

February 5, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Deprecations”](#)

[Previous Backward Compatibility](#) [Previous: Backward Compatibility](#)

[Next How To Get Your Pull Request Reviewed?](#) [Next: How To Get Your Pull Request Reviewed?](#)

How To Get Your Pull Request Reviewed?

In this article

[Table of Contents](#)

- [Create the smallest reasonable PRs](#)
- [Share relevant context:](#)
- [Make your PR exciting](#)
- [Show your work](#)
- [Review the work of others](#)
- [Reduce risk with clarity](#)
- [Follow the attention](#)

[↑ Back to top](#)

Sometimes we publish a Pull Request and no one [reviews](#) our work. What to do?

Attracting a review largely isn't about the code – it is about making the reviewing easy.

If you published a Pull Request that isn't getting any comments or reviews, try one of the strategies used by core contributors:

[**Create the smallest reasonable PRs**](#)

Approving a 2000-line-long PR takes months and feels overwhelming.

Approving a 50-line long PR takes days or hours and feels easy.

Large batches slow you down. Ship your work in small chunks to merge more and learn faster.

[**Share relevant context:**](#)

Clarify:

- * What problem are you solving?
- * How does your PR solve it?
- * What feedback do you need?
- * What's out of scope?
- * What's unintuitive?
- * How to test?

Summarize any related issues and PRs.

It's easier than asking others to go and figure it out.

[**Make your PR exciting**](#)

All contributions are competing for attention. Make your stand out.

The easiest way? Say why it matters:

-  A new react hook to get data
 `useEntityRecord`: get data with 10x less boilerplate

Then prove it with code examples, visuals, and screencasts.

Show your work

Post a link to your PR in related issues & PRs.

Ping commenters of related issues, previous committers, and tech leads.

Bring it up on the #core-editor channel of the WordPress.org slack. The easiest way to get feedback is to speak out during the [open floor section](#) of the weekly [Core Editor meeting](#).

Assign relevant labels, milestones, and projects (or ask someone).

Review the work of others

It's the easiest way to get on others' radar.

Look up the PRs of commenters of related issues, previous committers, and tech leads. Then review them.

Is their work unfamiliar? Do:

- Take some time to understand it
- Propose a pair programming session
- Skip, go for the next PR

Reduce risk with clarity

Risk adds friction – an approval can backfire later.

Clarity is like grease. Clearly document:

- What risks are involved? Why take them?
- Why is this PR the best solution?
- How can the risk be minimized?
- What else has been tried?

Follow the attention

Some PRs naturally get more traction than others.

Double down on these.

Some Issues are more topical than others (e.g. those listed in the goals for an upcoming release) and thus will garner more attention. By focusing on these it will be easier to attract reviewers.

How to get there quickly? Help with an active project from the WordPress roadmap

First published

May 5, 2022

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: How To Get Your Pull Request Reviewed?"](#)

[Previous Deprecations](#) [Previous: Deprecations](#)

[Next Design Contributions](#) [Next: Design Contributions](#)

Design Contributions

In this article

[Table of Contents](#)

- [Discussions](#)
- [How can designers contribute?](#)
- [Principles](#)
 - [Goal of Gutenberg](#)
 - [Why](#)
 - [Vision](#)

[↑ Back to top](#)

A guide on how to get started contributing design to the Gutenberg project.

[Discussions](#)

The [Make WordPress Design blog](#) is the primary spot for the latest information around WordPress Design Team: including announcements, product goals, meeting notes, meeting agendas, and more.

Real-time discussions for design take place in the `#design` channel in [Make WordPress Slack](#) (registration required). Weekly meetings for the Design team are on Wednesdays at 19:00UTC.

[How can designers contribute?](#)

The Gutenberg project uses GitHub for managing code and tracking issues. The main repository is at: <https://github.com/WordPress/gutenberg>.

If you'd like to contribute to the design or front-end, feel free to contribute to tickets labeled [Needs Design](#) or [Needs Design Feedback](#). We could use your thoughtful replies, mockups, animatics, sketches, doodles. Proposed changes are best done as minimal and specific iterations on the work that precedes it so we can compare.

The [WordPress Design team](#) uses [Figma](#) to collaborate and share work. If you'd like to contribute, join the [#design channel](#) in [Slack](#) and ask the team to set you up with a free Figma account. This will give you access to a helpful [library of components](#) used in WordPress.

Principles

This section outlines the design principles and patterns of the editor interface—to explain the background of the design, inform future improvements, and help people design great blocks.



The Gutenberg logo was made by [Cristel Rossignol](#), and is released under the GPL license. [Download the SVG logo](#).

Goal of Gutenberg

Gutenberg's all-encompassing goal is a post- and page-building experience that makes it easy to create rich layouts. The block editor was the first product launched following this methodology for working with content.

From the [kickoff post](#):

The editor will endeavor to create a new page and post building experience that makes writing rich posts effortless, and has “blocks” to make easy what today might take shortcodes, custom HTML, or “mystery meat” embed discovery.

We can extract a few key principles from this:

- **Authoring rich posts is a key strength of WordPress.**
- **Blocks will unify features and types of interaction under a single interface.** Users shouldn't have to write shortcodes, custom HTML, or paste URLs to embed. Users only need to learn how the block works in order to use all of its features.
- **Make core features more discoverable**, reducing hard-to-find “Mystery meat.” WordPress supports a large number of blocks and 30+ embeds. Let's increase their visibility.

Why

One thing that sets WordPress apart from other systems is that it allows users to create as rich a post layout as they can imagine — as long as they know HTML and CSS and build a custom theme.

Gutenberg reshapes the editor into a tool that allows users to write rich posts and build beautiful layouts in a few clicks — no technical knowledge needed. WordPress will become a powerful and flexible content tool that's accessible to all.

Vision

Gutenberg wants to make it easier to author rich content. This means ensuring good defaults, bundling advanced layout options into blocks, and making the most important actions immediately available. Authoring content with WordPress should be accessible to anyone.

Everything on a WordPress website becomes a block: text, images, galleries, widgets, shortcodes, and even chunks of custom HTML, whether added by plugins or otherwise. Users will only have to learn a single interface — the block interface.

All blocks are created equal. They all live in the same inserter interface. Recency, search, tabs, and grouping ensure that the most-used blocks are within easy reach.

Drag-and-drop is secondary. For greater accessibility and platform compatibility, drag-and-drop interactions are used as an additive enhancement on top of explicit actions like click, tab, and space.

Placeholders are key. If a block can have a neutral placeholder state, it should. An image placeholder block shows a button to open the media library, and a text placeholder block shows a

writing prompt. By embracing placeholders we can predefine editable layouts, so all users have to do is fill in the blanks.

Direct manipulation is intuitive. The block interface allows users to manipulate content directly on the page. Plugin and theme authors will support and extend this experience by building their own custom blocks.

Code editing shouldn't be necessary for customization. Customizing traditionally required complicated markup, and complicated markup is easy to break. With Gutenberg, customizing becomes more intuitive — and safer. A developer will be able to provide custom blocks that directly render portions of a layout (a three column grid of features, for instance) and clearly specify what can be directly edited by the user. That means the user can update text, swap images, reduce the number of columns, without having to ask a developer, or worrying about breaking things.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Design Contributions”](#)

[Previous How To Get Your Pull Request Reviewed?](#) [Previous: How To Get Your Pull Request Reviewed?](#)

[Next Blocks are the Interface](#) [Next: Blocks are the Interface](#)

Blocks are the Interface

In this article

[Table of Contents](#)

- [Building blocks](#)
 - [The primary interface for a block is the content area of the block.](#)
 - [The block toolbar is the place for critical options that can't be incorporated into placeholder UI.](#)
 - [The Settings Sidebar should only be used for advanced, tertiary controls.](#)

[↑ Back to top](#)

At the core of Gutenberg lies the concept of the block. From a technical point of view, blocks both raise the level of abstraction from a single document to a collection of meaningful elements, and they replace ambiguity— inherent in HTML—with explicit structure.

From a user perspective, blocks allow any kind of content, media, or functionality to be directly added to their site in a more consistent and usable way. The “add block” button gives the user

access to an entire library of options all in one place, rather than having to hunt through menus or know shortcodes.

But most importantly, Gutenberg is built on the principle of *direct manipulation*, which means that the primary options for how an element is displayed are controlled *in the context of the block itself*. This is a big shift from the traditional WordPress model, where options that were often buried deep in layers of navigation menus controlled the elements on a page through indirect mechanisms.

So, for example, a user can add an image, write its caption, change its width and layout, add a link around it, all from within the block interface in the canvas. The same principle should apply to more complex blocks, like a “navigation menu”, with the user being able to add, edit, move, and finalize the full presentation of their navigation.

- Users only need to learn one interface — the block — to add and edit everything on their site. Users shouldn't have to write shortcodes, custom HTML, or understand hidden mechanisms to embed content.
- Gutenberg makes core features more discoverable, reducing hard-to-find “Mystery meat.” WordPress supports a large number of blocks and 30+ embeds. Let's increase their visibility.

Building blocks

What does this mean for designers and developers? The block structure plus the principle of direct manipulation mean thinking differently about how to design and develop WordPress components. Let's take another look at the architecture of a block:

Blo

Block Mover



The primary interface for a block is the content area of the block.

The placeholder content in the content area of the block can be thought of as a guide or interface for users to follow a set of instructions or “fill in the blanks” (more on placeholders later). Since the content area represents what will actually appear on the site, interaction here hews closest to the principle of direct manipulation and will be most intuitive to the user. This should be thought of as the primary interface for adding and manipulating content and adjusting how it is displayed.

The block toolbar is the place for critical options that can't be incorporated into placeholder UI.

Basic block settings won’t always make sense in the context of the placeholder / content UI. As a secondary option, options that are critical to the functionality of a block can live in the block toolbar. The block toolbar is one step removed from direct manipulation, but is still highly contextual and visible on all screen sizes, so it is a great secondary option.

The Settings Sidebar should only be used for advanced, tertiary controls.

The Settings Sidebar is not visible by default on a small / mobile screen, and may also be collapsed even in a desktop view. Therefore, it should not be relied on for anything that is necessary for the basic operation of the block. Pick good defaults, make important actions available in the block toolbar, and think of the sidebar as something that only power users may discover.

First published

March 10, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Blocks are the Interface”](#)

[Previous Design Contributions](#) [Previous: Design Contributions](#)

[Next Documentation Contributions](#) [Next: Documentation Contributions](#)

Documentation Contributions

In this article

Table of Contents

- [Discussions](#)
- [Documentation types](#)
- [Block editor handbook process](#)
 - [Handbook structure](#)
 - [Templates](#)
 - [Update a document](#)
 - [Create a new document](#)

- [Documenting packages](#)
- [Using links](#)
- [Code examples](#)
- [Callout notices](#)
- [Editor config](#)
- [Video embeds](#)
- [Resources](#)

[↑ Back to top](#)

A guide on how to get started contributing documentation to the Gutenberg project.

Discussions

The [Make WordPress Docs blog](#) is the primary spot for the latest information around WordPress documentation, including announcements, product goals, meeting notes, meeting agendas, and more.

Real-time discussions for documentation take place in the #docs channel in [Make WordPress Slack](#) (registration required). Weekly meetings for the Documentation team are on Tuesdays at 14:00UTC.

The Gutenberg project uses GitHub for managing code and tracking issues. The main repository is at: <https://github.com/WordPress/gutenberg>. To find documentation issues to work on, browse [issues with documentation label](#).

Documentation types

There are two major sets of documentation for the Gutenberg project:

1. [User documentation](#) is information on how to use the Editor as an author publishing posts. For contributing to user docs, follow the docs blog or ask in the #docs Slack channel to understand the current priorities.
2. [Block editor handbook](#) is everything related to the Gutenberg project including: developing, extending, and—what you are reading right now—contributing specific to Gutenberg.

The rest of this document covers contributing to the block editor handbook.

Block editor handbook process

The block editor handbook is a mix of markdown files in the /docs/ directory of the [Gutenberg project repository](#) and generated documentation from the packages.

An automated job publishes the docs every 15 minutes to the [block editor handbook site](#).

See [the Git Workflow](#) documentation for how to use git to deploy changes using pull requests. Additionally, see the [video walk-through](#) and the accompanying [slides for contributing documentation to Gutenberg](#).

[Handbook structure](#)

The handbook is organized into four sections based on the functional types of documents. [The Documentation System](#) does a great job explaining the needs and functions of each type, but in short, they are:

- **Getting started tutorials** – full lessons that take learners step by step to complete an objective, for example the [create a block tutorial](#).
- **How-to guides** – short lessons specific to completing a small specific task, for example [how to add a button to the block toolbar](#).
- **Reference guides** – API documentation, purely functional descriptions,
- **Explanations** – longer documentation focused on learning, not a specific task.

[Templates](#)

A [how-to guide template](#) is available to provide a common structure to guides. If starting a new how-to guide, copy the markdown from the template to get started.

The template is based on examples from The Good Docs Project. See their [template repository](#) for additional examples to help you create quality documentation.

[Update a document](#)

To update an existing page:

1. Check out the Gutenberg repository.
2. Create a branch to work, for example `docs/update-contrib-guide`.
3. Make the necessary changes to the existing document.
4. Commit your changes.
5. Create a pull request using the [\[Type\] Developer Documentation](#) label.

[Create a new document](#)

To add a new document requires a working JavaScript development environment to build the documentation, see the [JavaScript build setup documentation](#):

1. Create a Markdown file in the `docs` folder, use lower-case, no spaces, if needed a dash separator, and `.md` extension.
2. Add content using markdown notation. All documents require one and only `h1` tag.
3. Add document entry to the `toc.json` hierarchy. See existing entries for format.
4. Run `npm run docs:build` to update `manifest.json`.
5. Commit `manifest.json` with other files updated.

If you forget to run, `npm run docs:build` your PR will fail the static analysis check since the `manifest.json` file is an uncommitted local change that must be committed.

[Documenting packages](#)

Package documentation is generated automatically by the documentation tool by pulling the contents of the `README.md` file located in the root of the package. Sometimes, however, it is preferable to split the contents of the `README` into smaller, easier-to-read portions.

This can be accomplished by creating a `docs` directory in the package and adding `toc.json` file that contains references other markdown files also contained in the `docs` directory. The `toc.json` file should contain an array of pages to be added as sub-pages of the main `README` file. The formatting follows the [manifest.json](#) file that is generated automatically.

In order for these pages to be nested under the main package name, be sure to set the `parent` property correctly. See the example below that adds child pages to the [@wordpress/create-block section](#).

```
[  
  {  
    "title": "@wordpress/create-block External Template",  
    "slug": "packages-create-block-external-template",  
    "markdown_source": "../packages/create-block/docs/external-template.md",  
    "parent": "packages-create-block"  
  }  
]
```

[Using links](#)

It's likely at some point, you'll want to link to other internal documentation pages. It's worth emphasizing all documents can be browsed in different contexts:

- Block editor handbook
- GitHub website
- npm website

To create links that work in all contexts, you must use absolute path links without the 'https://github.com/WordPress/gutenberg' prefix. You can reference files using the following patterns:

- `/docs/*.md`
- `/packages/*/README.md`
- `/packages/components/src/**/README.md`

This way, they will be properly handled in all three aforementioned contexts.

Use the full directory and filename from the Gutenberg repository, not the published path; the Block Editor Handbook creates short URLs—you can see this in the tutorials section. Likewise, the `readme.md` portion is dropped in the handbook but should be included in the links.

An example, the link to this page is: `/docs/contributors/documentation/README.md`

Note: The usual link transformation is not applied to links in callouts. See below.

[Code examples](#)

The code example in markdown should be wrapped in three tick marks ````` and should additionally include a language specifier. See this [GitHub documentation around fenced code blocks](#).

A unique feature to the Gutenberg documentation is the `codetabs` toggle, this allows two versions of code to be shown at once. This is used for showing both `JSX` and `Plain` code samples.

Here is an example `codetabs` section:

```
\{\%\ codetabs \%\}
\{\%\ JSX \%\}
```js
// JSX code here
```
\{\%\ Plain \%\}
```js
// Plain code here
```
\{\%\ end \%\}
```

The preferred format for code examples is JSX. This should be the default view. The example placed first in source will be shown as the default.

Note: It is not required to include plain JavaScript code examples for every guide. The recommendation is to include plain code for beginner tutorials or short examples, but the majority of code in Gutenberg packages and across the larger React and JavaScript ecosystem are in JSX that requires a build process.

Callout notices

The Block Editor handbook supports the same [notice styles as other WordPress handbooks](#). However, the shortcode implementation is not ideal with the different locations the block editor handbook documentation is published (npm, GitHub).

The recommended way to implement in markdown is to use the raw HTML and `callout` `callout-LEVEL` classes. For example:

```
<div class="callout callout-info">This is an **info** callout.</div>
```

The following classes are available: `info`, `tip`, `alert`, `warning`

This is a `tip` callout.

This is an `info` callout.

This is an `alert` callout.

This is a `warning` callout.

Note: In callout notices, links also need to be HTML `` notations.

The usual link transformation is not applied to links in callouts.

For instance, to reach the Getting started > Create Block page, the URL in GitHub is <https://developer.wordpress.org/docs/getting-started/devenv/get-started-with-create-block.md> and will have to be hardcoded for the endpoint in the Block Editor Handbook as <https://developer.wordpress.org/block-editor/getting-started/create-block/> to link correctly in the handbook.

Editor config

You should configure your editor to use Prettier to auto-format markdown documents. See the [Getting Started documentation](#) for complete details.

An example config for using Visual Studio Code and the Prettier extensions:

```
"[ [markdown] ]": {  
    "editor.defaultFormatter": "esbenp.prettier-vscode",  
    "editor.formatOnSave": true  
},
```

Depending on where you are viewing this document, the brackets may show as double. The proper format is just a single bracket.

[Video embeds](#)

Videos in the Block Editor Handbook need to be hosted on the [WordPress YouTube channel](#) as unlisted videos. This process requires additional permissions. Reach out in the #marketing Slack channel for assistance.

Once the video has been uploaded to YouTube, retrieve the video embed link. It should look something like this:

<https://www.youtube.com/embed/nrut8SfXA44?si=YxvmHmAoYx-BDCog>

Then, place the following code where you want the video to be embedded in the documentation. Update the embed link and video title accordingly.

```
<iframe width="960" height="540" src="[Video embed link]" title="[Video ti
```

Videos should have an aspect ratio of 16 : 9 and be filmed at the highest resolution possible.

[Resources](#)

- [Copy Guidelines](#) for writing instructions, documentation, or other contributions to the Gutenberg project.
- [Tone and Voice Guide](#) from WordPress Documentation.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Documentation Contributions”](#)

[Previous](#) [Blocks are the Interface](#) [Previous: Blocks are the Interface](#)
[Next](#) [Copy Guidelines](#) [Next: Copy Guidelines](#)

Copy Guidelines

In this article

Table of Contents

- [Longer Text](#)
- [Bulleted Lists](#)
- [UI Descriptions](#)
- [Error Messaging](#)

[↑ Back to top](#)

[Longer Text](#)

Guidelines for writing multi-line/step instructions or narrative introductions/orientation to pages or features.

This will obviously vary quite a lot depending on the context, but here are some general tips:

ONE: Contractions are your friends!

They're more conversational, and a simple way to make text sound friendlier and less formal. (And they save a bit of space as well: a win-win.)

TWO: Cut phrases that inflate your word count without actually adding meaning.

This happens frequently in two specific instances. First, when writing in the passive voice:

This block can be used to display single images.

Any time you see phrases like “can be” or “is used”: halt. You’re writing in the passive voice. Try going active for a snappier (and shorter) sentence:

This block displays single images.

Second, this happens when we hedge instead of making declarative statements:

The gallery block can help you display multiple images in an elegant layout.

Does it or doesn’t it? We’re making this software: we’re allowed to be declarative about what it is and does:

The gallery block displays multiple images in an elegant layout.

We also all do this a lot with the phrase “allows you to.”

Preformatted text allows you to keep your tabs and line breaks.

Features don’t allow anyone to do anything; they’re just tools that do specific things to achieve an end. Just say what they do:

Preformatted text preserves your tabs and line breaks.

The more direct sentences are almost always clearer. Scan your copy for the words “can,” “be,” “might,” “allows you to,” and “helps”—they’re the most common culprits, and looking for those words specifically is a way to locate phrasing you can tighten up.

THREE: Beware of “simple,” “easy,” and “just.”

It is not for us to decide what is simple: it’s for the user to decide. If we say something is easy and the user doesn’t have an easy experience, it undermines their trust in us and what we’re building. The same goes for “just”—many of us know to avoid “simple,” but still use “just” all the time. “Just click here.” “Just enter your username.” It’s the same thing: it implies that something will be no big deal, but we can’t know what the user will find to be a big deal.

It’s also safer and more helpful to be specific. “Easy” and “simple” are shorthand for explanations that we haven’t written; whenever you see them, take a minute to think about what they’re standing in for. Maybe “It’s easy to add a block by hitting ‘enter’” really means “You can add more content to the page without taking your hands off the keyboard.” Great! Say the specific thing instead of relying on “easy.”

This isn’t to say that you should banish these words from your vocabulary. You might want to write a tooltip describing how the cover image block now requires less configuration, or an email about how we’re building a tool for quick creation of custom blocks, and you could legitimately say that the cover image block has been simplified or that we’re working to make custom block creation easier—there, the terms are descriptive and relative. But be on the lookout for ways you might be using (or overusing) them to make absolute claims that something is easy or simple, and use those as opportunities to be more specific and clear.

FOUR: Look out for “we.”

Any time text or instructions uses “we” a lot, it means the focus of the text is on the people behind the software and not the people using the software. Sometimes that’s what you actually want—but it’s usually not. The focus should typically be on the user, what they need, and how they benefit rather than “what we did” or “what we want.”

We’re the only ones that care about what we did or want; the user just wants software that works. If you see a lot of “we”s, think about whether you should reframe what you’re writing to focus on the benefits to and successes of the user.

Bulleted Lists

Guidelines for (duh) writing bulleted lists.

ONE: Keep sentence structures parallel across all bullets.

Parallel structure makes lists easier to read quickly—their predictability takes some cognitive load off the reader.

GOOD:

What can you do with this block? Lots of things!

- Add a quote.

- Highlight a link.
- Display multiple images.
- Create a bulleted list.

Every bullet is a full sentence, and ends with a period. (If your list is a bunch of one- or two-word items, those can often just turn into a single regular sentence—easier to read, and space-saving.) Every line begins with a verb that tells the user what the block can do. The subject of the sentence is always the user.

A user can absorb this list quickly because once they read the first item, they understand how to read the rest and know what information they'll find.

LESS GOOD:

What can you do with this block? Lots of things!

- You can add a quote.
- Highlighting a link you love.
- It displays multiple images. Nice for galleries!
- Bulleted lists

Here, every line has different phrasing (some start with a verb, some with a noun) and the subject of the sentence changes (sometimes it's you, sometimes it's the block). Some lines have added description, some don't. There's an incomplete sentence, and punctuation is inconsistent.

Reading this list takes more work because the reader has to parse each bullet anew. They can't assume each bullet will contain similar information.

Note: this doesn't mean every bullet has to be super short and start with an action verb! "Predictable" doesn't have to mean "simple." It just means that each bullet should have the same sentence structure. This list would also be fine:

What can you do with this block? Lots of things!

- Try adding a quote. Sometimes someone else said things best!
- Use it to highlight a link you love—sharing links is the currency of the internet.
- Create a gallery that displays multiple images, and show off your best photos.

Here, each bullet starts with a more user-focused verb and includes a piece of supplemental information for more interest. The punctuation varies a bit, which keeps the lines from feeling too formulaic, but since the basic structure of each is the same, they remain easy to read.

TWO: When in doubt, start with a verb. (But not always the same verb.)

Do you have to start with a verb? No. But if you're at a loss, you usually can't go wrong with a verb (especially since bulleted lists are often describing a series of actions or possible actions).

In a simple list that's meant to be purely instructional (e.g., in UI copy where you just need the user to make a decision), it might be fine to start every bullet with the same verb:

To continue, choose an action:

- Add a simple text block.
- Add a pullquote block.

- Add an image block.

If your list is more persuasive (e.g., trying to convince someone to use a feature by listing its benefits) or includes multi-step instructions, you'll want to vary your verbs to keep the reader engaged with more interesting language, as in the example above:

What can you do with this block? Lots of things!

- Try adding a quote. Sometimes someone else said things best!
- Use it to highlight a link you love—sharing links is the currency of the internet.
- Create a gallery that displays multiple images, and show off your best photos.

These aren't hard-and-fast rules—you might choose the use the same verb in a persuasive list to be more focused and powerful, for example. But they're good starting places for solid lists.

THREE: When something's clearly a list, you don't have to tell us it's a list.

GOOD:

What can you do with this block? Lots of things!

- Add a quote.
- Highlight a link you love.
- Display multiple images.

LESS GOOD:

What can you do with this block? Lots of things! Here are some examples of ways you can use it.

- You can add a quote.
- Highlighting a link you love.
- It displays multiple images. Nice for galleries!

Find the balance between being as clear as possible and trusting a user. On one hand, we know that people don't always read instructions; on the other, redundancy can make the user feel like we think they're stupid.

FOUR: Bold is sometimes your friend.

Use it to focus readers on the key information in a bulleted list. This is especially useful when your bullets include some supplemental but ultimately secondary information.

“Key information” is, well, key: bold draws the eye, so stick to the most vital piece of information in a given bullet:

What can you do with this block? Lots of things!

- Try adding a **quote**. Sometimes someone else said things best!
- Use it to highlight a **link** you love—sharing links is the currency of the internet.
- Create a **gallery** that displays multiple images, and show off your best photos.

On the flipside, bolding too many things creates visual confusion:

What can you do with this block? Lots of things!

- Try adding a **quote**. Sometimes someone else said things best!
- Use it to highlight a **link** you love—sharing **links** is the currency of the internet.
- Create a **gallery** that displays **multiple images**, and show off your best **photos**.

When lists are short and basic, don't bother—bolding just adds busy-ness.

What can you do with this block? Lots of things!

- Add a **quote**.
- Highlight a **link**.
- Display multiple **images**.

The lack of words creates its own focus; you don't have to add any more.

UI Descriptions

Guidelines for writing one-line feature descriptions, or short descriptions to clarify options.

ONE: Clarity above all!

If the user doesn't understand what using a particular option will result in, it doesn't matter how clever your pun is. Wordplay and idioms are frequently unclear, and easily misunderstood. If you use them at all, they should be as supplemental information—never to explain the main idea—and they should be something you're fairly certain will be understandable to a pretty wide range of people.

TWO: Refer back to section one, and look out for those bulk-adding phrases.

Active voice is typically the better way to go, and cutting out the bulky phrasing is particularly important when you've got limited space and you need people to be able to make decisions and act. Often you can shorten a UI instruction phrase to be both shorter and clearer:

When you click X, Y happens.

vs.

Click X to do Y.

While it can feel like adding the extra words helps walk a user through the product, the extra words just serve to obscure the point being communicated:

When you click the “settings” button, the pop-up will display the advanced settings that are available.

vs.

Click “settings” to access the advanced settings.

Similar phrases are “Once you do X...” or “If you want to do X...” Sometimes there are decision points where “If you want to do X...” is entirely appropriate because there are different paths the

user can take based on their goal. But, we often use it to mean “Here is a thing you can do,” which you can express more simply as: “To do X...”

THREE: Be specific.

When an action depends on the user having completed some prior action, be specific about what’s required and what happens next. We often default to “when you’re ready.” Ready for what? Be specific about whatever the prerequisites are.

“When you’re ready” can mean:

- When you want to add another block”
- When you’re satisfied with your post”
- After you’ve finished proofreading your post”
- When you’d like to add a featured image”
- After you’ve configured all the settings”

And when something means everything, it actually means nothing. The more specific instructions are, the more useful they are, and the more trust the person following them will have in the product.

FOUR: This is still writing. It should have personality and interest.

Clarity above all, yes, and space is often limited here—but UI text can still be interesting to read.

Single lines of description can still be complete sentences.

List. Numbered or bulleted.

vs.

Add a list, either numbered or bulleted.

You can still use contractions.

Add a list. We will provide formatting options.

vs.

Add a bulleted list—we’ll give you some formatting options.

You can still use punctuation—em dashes, colons, semicolons—to control the flow of your words, link ideas, and create pauses.

List. Numbered or bulleted.

vs.

Add a list—numbered or bulleted. Your choice!

You can still try to avoid jargon in favor of plain language.

Add unordered or ordered list.

vs.

Add a list, either numbered or bulleted.

(And because it bears repeating: no wordplay, please! “Personality” can—and in UI instructions, should—be subtle. We’re talking about text that sounds like it was said by a human being, not forced attempts at whimsy.)

FIVE: Pay attention to capitalization.

When it comes to headlines and subheads, there are two ways to capitalize:

In Title Case, the First Letter of Almost Every Word Is Capitalized

In sentence case, only the first letter of the line is capitalized

Feature names and dashboard sections typically use title case (think “Site Stats” or “Recently Published”), whereas feature labels typically use sentence case (like “Show buttons on” or “Comment Likes are,” where “Likes” is capitalized because it’s the feature name, but the overall label is using sentence case).

When you’re looking at a full page of UI copy, make sure you’re being consistent across all of it, and that all similar kinds of copy—headlines, tooltips, buttons, etc.—are using the same case.

Error Messaging

Guidelines for writing error messages that are understandable and useful.

ONE: Don’t ignore voice/tone in error messaging—they communicate a lot.

Voice and tone can say as much as the individual words themselves. Error messages have to convey a significant amount of information and usually need to be fairly short, but try not to sacrifice tone, or to go too far in either a negative or positive direction.

Let’s say someone’s trying to publish a post, but their user role doesn’t allow them to do that. Here are some ways we could—but should not—communicate that:

Your user role is incorrect.

Here, we sound distant and uncaring.

Stop! You do not have permission to do this.

Here, we sound unnecessarily alarmist and stern.

Oopsie, we can’t let you do that!

Here, we sound too cute.

We can stay direct, positive, and friendly, even in error messages. How? With tips two through four!

TWO: Whenever possible, offer a path to resolution.

A good error message doesn’t just alert someone to the fact that something is wrong.

Your user role is incorrect.

Okay, fine. Why does that matter? What do I do about it? How does this message help me? I need to know why my user role matters, and how to get the role I need so I can complete the action I want to complete. An error message that doesn't provide any instruction leaves the user without a path forward; they can't avoid repeating the action that led to the error if we don't tell them now.

THREE: Don't lean on jargon to cut down on words when space is tight.

Your user role is incorrect. Contact a site administrator.

Maybe we're getting somewhere here: now I know there's something I can do about things, which is good.

Then again, maybe we're not: I still don't know what my role is, or why it matters. Also, now I'm not sure what a site administrator is, who mine is, or how to contact them.

All the information in this error message is technically entirely correct, but that doesn't mean it communicates anything useful. If the goal is understanding and resolution, technical accuracy doesn't always get us there.

"Your account does not have permission to publish posts" doesn't use the language of the user roles UI, but it does explain what's gone wrong and I can understand it even if I don't know what a user role is. And since I understand, I'm also better placed to understand the resolution, even if the message ended here: I can see that I need to get permission.

Consistency with existing UI language is great, but not when it gets in the way of understanding.

FOUR: Don't assume people understand where the error came from.

Your user role is incorrect.

It might seem obvious to us that the user got this message when they tried to publish something or change a setting that they don't have permission for. It might not be so obvious to the user: people click around a lot, especially when we're unsure how to do something, and we don't always remember what page or setting we were just looking at (or why!).

A good error message also includes some context that orients the user. "Your account does not have permission to publish posts" reminds them that they were trying to publish a post, and that that's the particular stumbling block that caused the error.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Copy Guidelines](#)

Triage

In this article

Table of Contents

- [Join the triage team](#)
- [Triage your first issues](#)
- [General triage process](#)
- [Closing issues](#)
- [Specific triages](#)
 - [Release specific triage](#)
 - [Design specific triage](#)

[↑ Back to top](#)

To keep the repository healthy, it needs to be triaged regularly. **Triage is the practice of reviewing existing issues and pull requests to make sure they're relevant, actionable, and have all the information they need.** Anyone can help triage, although you'll need to be a member of the triage team for the Gutenberg repository to modify an issue's labels or edit its title.

Besides this page, the [How to do triage on GitHub](#) tutorial is another great resource to get introduced to triage

[Join the triage team](#)

The triage team is an open group of people with a particular role of making sure triage is done consistently across the Gutenberg repo. There are various types of triage which happen:

- Regular self triage sessions done by members on their own time.
- Organised triage sessions done as a group at a set time. You can [review the meetings page](#) to find these triage sessions and appropriate slack channels.
- Focused triage sessions on a specific board, label or feature.

These are the expectations of being a triage team member:

- You are expected to do some triage even if it is self triage at least once a week.
- As you can, try to join organized triage sessions.
- If you join the triage team to focus on a specific label or board, the expectation is that your focus will be there. Please make this known to fellow triage team members.

If you would like to join this team, simply ask in #core-editor [Slack](#) at any time.

Triage your first issues

To start simply choose from one of these filtered lists below. Note: You can find most of these filters by selecting the “Sort” option from the [overall Issues page](#).

- **All Gutenberg issues without an assigned label.** Triaging by simply adding labels helps people focused on certain aspects of Gutenberg find relevant issues easier and start working on them.
- **All Gutenberg pull requests without an assigned label.** This requires a level of comfortability with code. For more guidance on which labels are best to use, please [review this section on labeling pull requests](#) for contributors. You can also always check with the person authoring the pull request to make sure the labels match what they are intending to do.
- **The least recently updated Gutenberg issues.** Triaging issues that are getting old and possibly out of date keeps important work from being overlooked.
- **All Gutenberg issues with no comments.** Triaging this list helps make sure all issues are acknowledged, and can help identify issues that may need more information or discussion before they are actionable.
- **The least commented on Gutenberg issues.** Triaging this list helps the community figure out what things might still need traction for certain proposals.
- **The most commented on Gutenberg issues.** If you feel comfortable chiming in and the conversation has stagnated, the best way to triage these kinds of issues is to summarize the discussion thus far and do your best to identify action items, blockers, etc. Triaging this list allows finding solutions to important and complex issues to move forward.
- You can also [create your own custom set of filters on GitHub](#). If you have a filter you think might be useful for the community, feel free to submit a PR to add it to this list.

General triage process

When triaging, either one of the lists above or issues in general, work through issues one-by-one. Here are some steps you can perform for each issue:

1. First **search for duplicates**. If the issue is duplicate, close it by commenting with “Duplicate of #” and add any relevant new details to the existing issue. (Don’t forget to search for duplicates among closed issues as well!).
2. If the **issue is missing labels, add some** to better categorize it (requires proper permissions given after joining the triage team). A good starting place when adding labels is to apply one of the labels prefixed [Type] (e.g. [Type] Enhancement or [Type] Bug) to indicate what kind of issue it is. After that consider adding more descriptive labels. If the issue concerns a particular core block, add one of the labels prefixed [Block]. Or if the issue affects a particular feature there are [Feature] labels. Finally, there are labels that affect particular interest areas, like Accessibility and Internationalization. You can view all possible labels [here](#).
3. If the **title doesn’t communicate the issue clearly enough, edit it for clarity** (requires proper permissions). Specifically, we’d recommend having the main feature the issue relates to in the beginning of the title ([example](#)) and for the title to generally be as succinct yet descriptive as possible ([example](#)).
4. If it’s a **bug report, test to confirm the report or add the Needs Testing label**. If there is not enough information to confirm the report, add the [Status] Needs More Info label and ask for the details needed. It’s particularly beneficial when a bug report has steps for reproduction so ask the reporter to add those if they’re missing.
5. **Remove the [Status] Needs More Info when is no longer needed**, for example if the author of the issue has responded with enough details.

6. **Close the inactive** [Status] Needs More Info issues with a note if the author didn't respond in 2+ weeks.
7. If there was a conversation on the issue but **no actionable steps identified, follow up with the participants to see what's actionable**. Make sure to @ each participant when responding in a comment.
8. If you feel comfortable triaging the issue further, then you can also:
 - Check that the bug report is valid by debugging it to see if you can track down the technical specifics.
 - Check if the issue is missing some detail and see if you can fill in those details. For instance, if a bug report is missing visual detail, it's helpful to reproduce the issue locally and upload a screenshot or GIF.
 - Consider adding the Good First Issue label if you believe this is a relatively easy issue for a first-time contributor to try to solve.

Commonly used labels

Generally speaking, the following labels are very useful for triaging issues and will likely be the ones you use the most consistently. You can view all possible labels [here](#).

Label	Reason
[Type] Bug	When an intended feature is broken.
[Type] Enhancement	When someone is suggesting an enhancement to a current feature.
[Type] Help Request	When someone is asking for general help with setup/implementation.
Needs Technical Feedback	When you see new features or API changes proposed.
Needs More Info	When it's not clear what the issue is or it would help to provide additional details.
Needs Testing	When a new issue needs to be confirmed or old bugs seem like they are no longer relevant.

Determining priority labels

If you have enough knowledge about the report at hand and feel confident in doing so, you can consider adding priority. Note that it's on purpose that no priority label infers a normal level.

Label	Reason
Priority: High	Fits one of the current focuses and is causing a major broken experience (including flow, visual bugs and blocks).
Priority: Low	Enhancements that aren't part of focuses, niche bugs, problems with old browsers.

Closing issues

Issues are closed for the following reasons:

- A PR and/or latest release resolved the reported issue.
- Duplicate of a current report.
- Help request that is best handled in the WordPress.org forums.
- An issue that's not able to be replicated.

- An issue that needs more information that the author of the issue hasn't responded to for 2+ weeks.
- An item that is determined as unable to be fixed or is working as intended.

Specific triages

Release specific triage

Here are some guidelines to follow when doing triage specifically around the time of a release. This is important to differentiate compared to general triage so problematic, release blocking bugs are properly identified and solutions are found.

- **If a bug is introduced in a release candidate (RC) and it's going to break many workflows**, add it to the version milestone and flag in the [#core-editor](#) channel in WordPress.org slack.
- **If a bug was introduced in the most recent version, and a next RC hasn't yet happened**, ideally the developers can push to fix it prior to RC! The amount of push for a fix should scale proportional to the potential of breakage. In this case, add to the RC milestone and, if deemed urgent, ping in the [#core-editor](#) channel in WordPress.org slack.
- **If a bug wasn't introduced in the most recent version**, do not add a milestone. Instead, use labels like [Priority] High if it's a pressing issue, and if needed you can call attention to it in the weekly core meetings.

Design specific triage

Along with the general triage flows listed previously, there are some specific additions to the flows for more design-centric triage for design minded folks participating in triage.

- PR testing and reviews: this should be your first stop for daily self triage.
- Label Needs Design Feedback: check if the issue does need design feedback and, if possible, give it. You can organize this by priority, project boards or by least commented. Once there are enough opinions, please remove this label and decide on next steps (ie adding the Needs Design label).
- Label Needs Design: Does it really need a design? Does this fit a focus? If it has a design mark as Needs Design Feedback to better categorize the issue.

Reminders:

- Ask for screenshots as needed.
- Ask for iterations and note any changes before merging.
- If the issue isn't in a board, check to see if it doesn't fit in a specific focus.
- If the issue/pull has not been prioritized yet, consider adding a priority label to help move the issue forward.

For more detailed information about weekly design triage and to join in, please [review this guide](#).

First published

June 16, 2020

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Triage”](#)

[Previous Copy Guidelines](#) [Previous: Copy Guidelines](#)
[Next Localizing Gutenberg](#) [Next: Localizing Gutenberg](#)

Localizing Gutenberg

[↑ Back to top](#)

The Gutenberg plugin is translated via the general plugin translation system (GlotPress) at <https://translate.wordpress.org>. Review the [GlotPress translation process documentation](#) for additional information.

To translate Gutenberg in your locale or language, [select your locale here](#) and translate *Development* (which contains the plugin’s string) and/or *Development Readme* (please translate what you see in the Details tab of the [plugin page](#)).

A Global Translation Editor (GTE) or Project Translation Editor (PTE) with suitable rights will process your translations in due time.

Language packs are automatically generated once 95% of the plugin’s strings are translated and approved for a locale.

The inclusion of Gutenberg into WordPress core means that more than 51% of WordPress installations running a translated WordPress installation have Gutenberg’s translated strings compiled into the core language pack as well.

First published

April 26, 2019

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Localizing Gutenberg”](#)

[Previous Triage](#) [Previous: Triage](#)
[Next Accessibility Testing](#) [Next: Accessibility Testing](#)

Accessibility Testing

In this article

Table of Contents

- [Getting started](#)
- [Keyboard testing](#)
- [Screen reader testing](#)
 - [NVDA with Firefox](#)
 - [VoiceOver with Safari](#)

[↑ Back to top](#)

This is a guide on how to test accessibility on Gutenberg. This is a living document that can be improved over time with new approaches and techniques.

[Getting started](#)

Make sure you have set up your local environment following the instructions on [Getting Started](#).

[Keyboard testing](#)

In addition to mouse, make sure the interface is fully accessible for keyboard-only users. Try to interact with your changes using only the keyboard:

- Make sure interactive elements can receive focus using Tab, Shift+Tab or arrow keys.
- Buttons should be activable by pressing Enter and Space.
- Radio buttons and checkboxes should be checked by pressing Space, but not Enter.

If the elements can be focused using arrow keys, but not Tab or Shift+Tab, consider grouping them using one of the [WAI-ARIA composite subclass roles](#), such as [toolbar](#), [menu](#) and [listbox](#).

If the interaction is complex or confusing to you, consider that it's also going to impact keyboard-only users.

[Screen reader testing](#)

According to the [WebAIM: Screen Reader User Survey #8 Results](#), these are the most common screen reader and browser combinations:

Screen Reader & Browser	# of Respondents	% of Respondents
JAWS with Chrome	259	21.4%
NVDA with Firefox	237	19.6%
NVDA with Chrome	218	18.0%
JAWS with Internet Explorer	139	11.5%
VoiceOver with Safari	110	9.1%

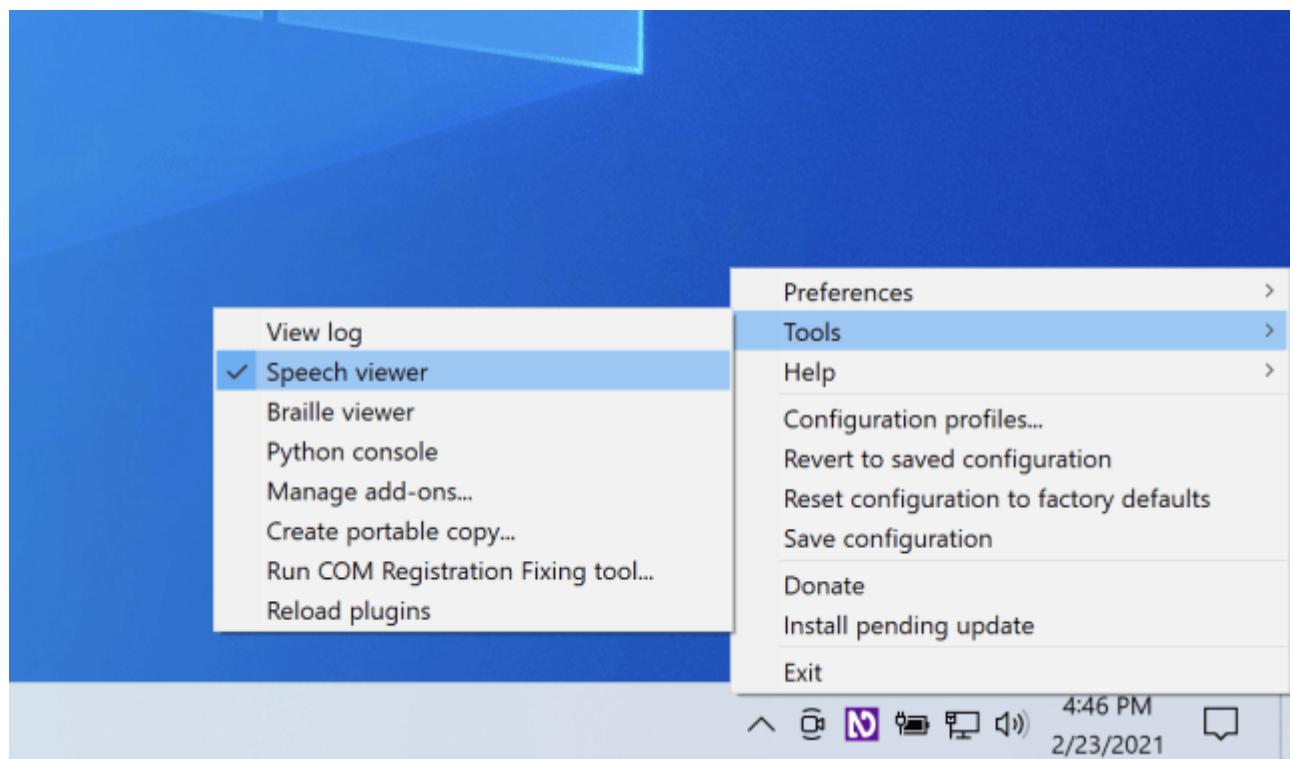
Screen Reader & Browser	# of Respondents	% of Respondents
JAWS with Firefox	71	5.9%
VoiceOver with Chrome	36	3.0%
NVDA with Internet Explorer	14	1.2%
Other combinations	126	10.4%

When testing with screen readers, try to use some of the combinations at the top of this list. For example, when testing with VoiceOver, it's preferable to use Safari.

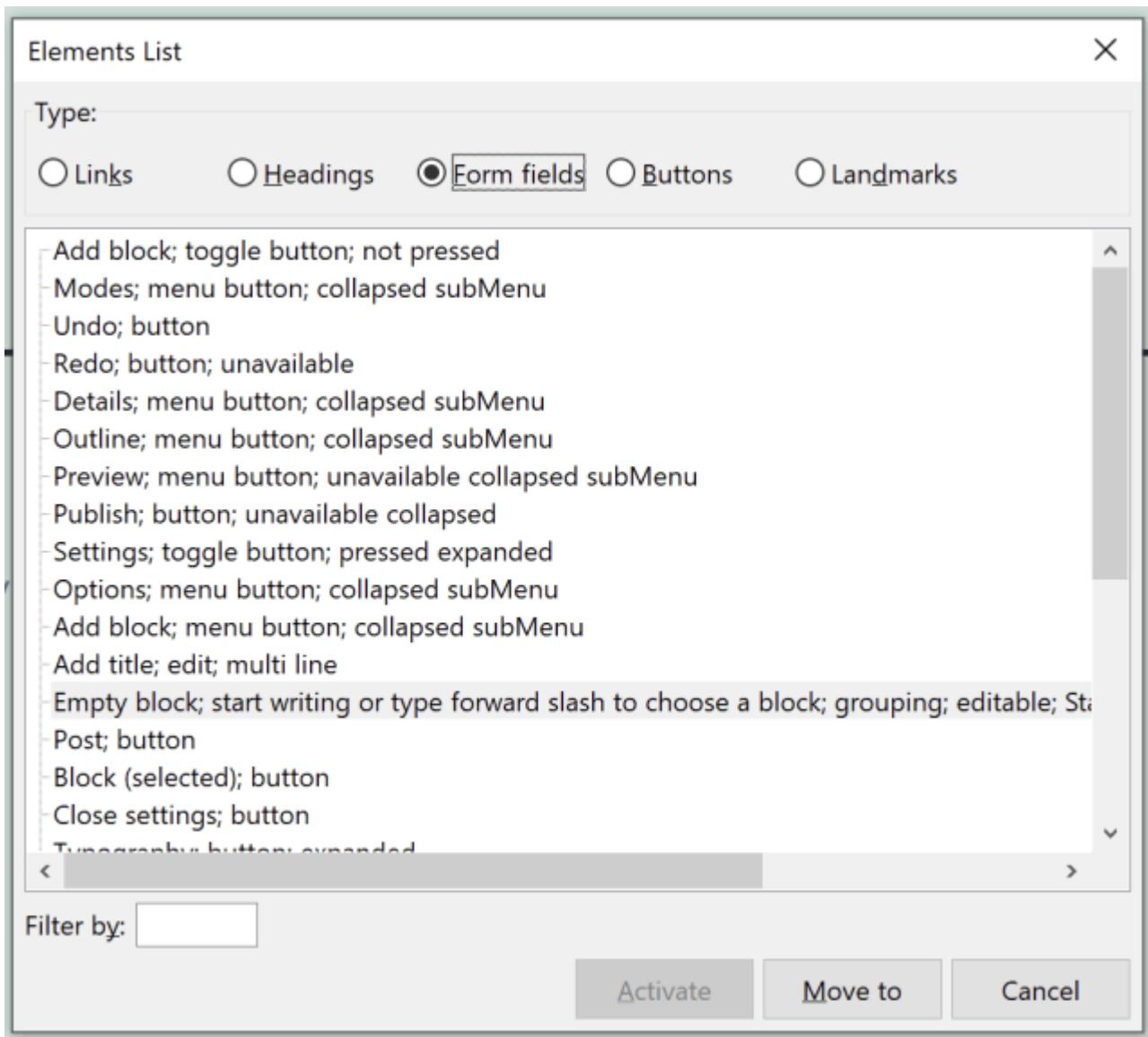
[NVDA with Firefox](#)

[NVDA](#) is a free screen reader for Windows and [the most popular one](#).

After installing it, you can activate NVDA by opening the app as you would do with other programs. An icon will appear on the System Tray where you have access to more options. It's recommended to enable the "Speech viewer" dialog so it's easier to demonstrate what's being announced by NVDA when you take screenshots.



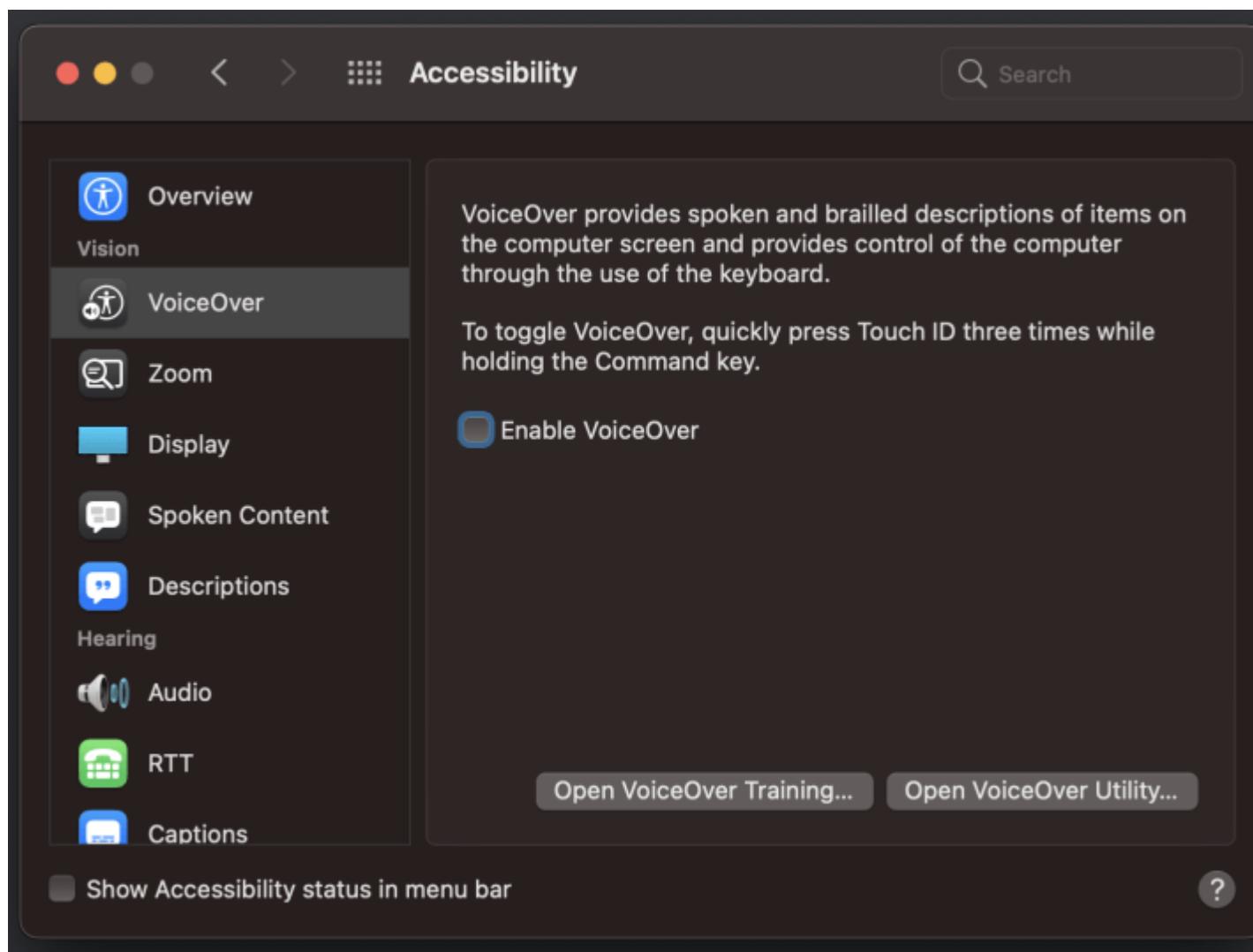
While in the Gutenberg editor, with NVDA activated, you can press **Insert+F7** to open the Elements List where you can find elements grouped by their types, such as links, headings, form fields, buttons and landmarks.



Make sure the elements have proper labels and prefer to navigate through landmarks and then use Tab and arrow keys to move through the elements within the landmarks.

VoiceOver with Safari

VoiceOver is the native screen reader on macOS. You can enable it on System Preferences > Accessibility > VoiceOver > Enable VoiceOver or by quickly pressing Touch ID three times while holding the Command key.



While in the Gutenberg editor, with VoiceOver activated, you can press **Control+Option+U** to open the Rotor and find more easily the different regions and elements on the page. This is also a good way to make sure elements are labelled correctly. If a name on this list doesn't make sense, it should be improved.

Form Controls

Editor top bar region
Add block toggle button
Modes collapsed button
Undo button
Redo dimmed button
Details collapsed button
Outline collapsed button
Preview dimmed collapsed button
Publish dimmed collapsed button
Settings toggle button
Options collapsed button
Editor content region
Add block group
Add block collapsed button
Add title Add title edit text
Empty block; start writing or type forward slash to...

Prefer to select a region or another larger area to begin with instead of individual elements on the Rotor so you can better test the navigation within that region.

Once you find the region you want to interact with, you can use Control+Option plus right or left arrow keys to move to the next or previous elements on the page. Then, follow the instructions that VoiceOver will announce.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Accessibility Testing”](#)

[Previous Localizing Gutenberg](#) [Previous: Localizing Gutenberg](#)

[Next Repository Management](#) [Next: Repository Management](#)

Repository Management

In this article

[Table of Contents](#)

- [Issues](#)
 - [Labels](#)
 - [Milestones](#)
 - [Triaging issues](#)
- [Pull requests](#)
 - [Code review](#)
 - [Design review](#)
 - [Merging pull requests](#)
 - [Closing pull requests](#)
- [Teams](#)
- [Projects](#)

[↑ Back to top](#)

This is a living document explaining how we collaboratively manage the Gutenberg repository. If you'd like to suggest a change, please open an issue for discussion or submit a pull request to the document.

This document covers:

- [Issues](#)
 - [Labels](#)
 - [Milestones](#)
 - [Triaging Issues](#)
- [Pull Requests](#)
 - [Code Review](#)
 - [Design Review](#)
 - [Merging Pull Requests](#)
 - [Closing Pull Requests](#)
 - [How To Get Your Pull Request Reviewed?](#)
- [Projects](#)

[Issues](#)

A healthy issue list is one where issues are relevant and actionable. *Relevant* in the sense that they relate to the project's current priorities. *Actionable* in the sense that it's clear what action(s) need to be taken to resolve the issue.

Any issues that are irrelevant or not actionable should be closed, because they get in the way of making progress on the project. Imagine the issue list as a desk: the more clutter you have on it, the more difficult it is to use the space to get work done.

Labels

All issues should have [one or more labels](#).

Workflow labels start with “Needs” and may be applied as needed. Ideally, each workflow label will have a group that follows it, such as the Accessibility Team for `Needs Accessibility Feedback`, the Testing Team for `Needs Testing`, etc.

[Priority High](#) and [Priority OMGWTFBBQ](#) issues should have an assignee and/or be in an active milestone.

Help requests or ‘how to’ questions should be posted in a relevant support forum as a first step. If something might be a bug but it’s not clear, the Support Team or a forum volunteer can help troubleshoot the case to help get all the right information needed for an effective bug report.

Here are some labels you might commonly see:

- [Good First Issue](#) – Issues identified as good for new contributors to work on. Comment to note that you intend to work on the issue and reference the issue number in the pull request you submit.
- [Good First Review](#) – Pull requests identified as good for new contributors who are interested in doing code reviews.
- [Needs Accessibility Feedback](#) – Changes that impact accessibility and need corresponding review (e.g. markup changes).
- [Needs Design Feedback](#) – Changes that modify the design or user experience in some way and need sign-off.
- [\[Type\] Bug](#) – An existing feature is broken in some way.
- [\[Type\] Enhancement](#) – Gutenberg would be better with this improvement added.
- [\[Type\] Plugin Interoperability](#) – Documentation of a conflict between Gutenberg and a plugin or extension. The plugin author should be informed and provided documentation on how to address.
- [\[Status\] Needs More Info](#) – The issue needs more information in order to be actionable and relevant. Typically this requires follow-up from the original reporter.

[Check out the label directory](#) for a listing of all labels.

Milestones

We put issues into [milestones](#) to better categorize them. Issues are added to milestones starting with `WordPress` and pull requests are added to milestones ending in `(Gutenberg)`.

Here are some milestones you might see:

- [WordPress X.Y](#): Tasks that should be done for future WordPress releases.
- [X.Y \(Gutenberg\)](#): PRs targeted for the Gutenberg Plugin X.Y release.
- [Future](#): this is something that is confirmed by everyone as a good thing but doesn’t fall into other criteria.

Triaging issues

To keep the issue list healthy, it needs to be triaged regularly. *Triage* is the practice of reviewing existing issues to make sure they’re relevant, actionable, and have all the information they need.

Anyone can help triage, although you'll need contributor permission on the Gutenberg repository to modify an issue's labels or edit its title.

See the [Triage Contributors guide](#) for details.

Pull requests

Gutenberg follows a feature branch pull request workflow for all code and documentation changes. At a high-level, the process looks like this:

1. Check out a new feature branch locally.
2. Make your changes, testing thoroughly.
3. Commit your changes when you're happy with them, and push the branch.
4. Open your pull request.
5. If you are a regular contributor with proper access, label and name your pull request appropriately (see below).

For labeling and naming pull requests, here are guidelines to consider that make compiling the changelog more efficient and organized. These guidelines are particularly relevant for regular contributors. Don't let getting the following right be a blocker for sharing your work – mistakes are expected and easy to fix!

- When working on experimental screens and features, apply the [Type] Experimental label instead of Feature, Enhancement, etc.
- When working on new features to technical packages (scripts, create-block, adding react hooks, etc), apply the [Type] New API label instead of Feature, Enhancement, etc.
- When fixing a bug or making an enhancement to an internal tool used in the project, apply the [Type] Build Tooling instead of Bugs, Enhancement, etc
- In pull request titles, instead of describing the code change done to fix an issue, consider referring to the actual bug being fixed instead. For example: instead of saying “Check for nullable object in component”, it would be preferable to say “Fix editor breakage when clicking the copy block button”.

Along with this process, there are a few important points to mention:

- Non-trivial pull requests should be preceded by a related issue that defines the problem to solve and allows for discussion of the most appropriate solution before actually writing code.
- To make it far easier to merge your code, each pull request should only contain one conceptual change. Keeping contributions atomic keeps the pull request discussion focused on one topic and makes it possible to approve changes on a case-by-case basis.
- Separate pull requests can address different items or todos from their linked issue, there's no need for a single pull request to cover a single issue if the issue is non-trivial.

Code review

Every pull request goes through a manual code review, in addition to automated tests. The objectives for the code review are best thought of as:

- Correct — Does the change do what it's supposed to?
- Secure — Would a nefarious party find some way to exploit this change?
- Readable — Will your future self be able to understand this change months down the road?
- Elegant — Does the change fit aesthetically within the overall style and architecture?

- Altruistic — How does this change contribute to the greater whole?

As a reviewer, your feedback should be focused on the idea, not the person. Seek to understand, be respectful, and focus on constructive dialog.

As a contributor, your responsibility is to learn from suggestions and iterate your pull request should it be needed based on feedback. Seek to collaborate and produce the best possible contribution to the greater whole.

Code reviews are encouraged by everyone who is willing to attempt one. If you review a pull request and are confident in the changes, approve it. If you don't feel totally confident it is ready for merging, add your review with a comment that says it should have another set of eyes on it before final approval. This can help filter out obvious bugs and simplify reviews for core members. Following the later reviews will also help improve your reviewing confidence in the future.

If you are not yet comfortable leaving a full review, try commenting on a PR. Questions about functionality or the reasoning behind a change are helpful too. You could also comment on changes to parts of the code you understand, without leaving a full review.

If you struggle with getting a review, see: [How To Get Your Pull Request Reviewed?](#)

Design review

If your pull request impacts the design/UI, you need to label appropriately to alert design. To request a design review, add the [Needs Design Feedback](#) label to your PR. If there are any PRs that require an update to the design/UI, please use the [Figma Library Update](#) label.

As a guide, changes that should be reviewed:

- A change based on a previous design, to confirm the design is still valid with the change.
- Anything that changes something visually.
- If you just want design feedback on an idea or exploration.

Merging pull requests

A pull request can generally be merged once it is:

- Deemed a worthwhile change to the codebase.
- In compliance with all relevant code review criteria.
- Covered by sufficient tests, as necessary.
- Vetted against all potential edge cases.
- Changelog entries were properly added.
- Reviewed by someone other than the original author.
- [Rebased](#) onto the latest version of the `trunk` branch.

The final pull request merge decision is made by the [@wordpress/gutenberg-core](#) team.

All members of the WordPress organization on GitHub have the ability to review and merge pull requests. If you have reviewed a PR and are confident in the code, approve the pull request and comment ping [@wordpress/gutenberg-core](#) or a specific core member who has been involved in the PR. Once they confirm there are no objections, you are free to merge the PR into trunk.

Most pull requests will be automatically assigned a release milestone, but please make sure your merged pull request was assigned one. Doing so creates the historical legacy of what code landed when, and makes it possible for all project contributors (even non-technical ones) to access this information.

Closing pull requests

Sometimes, a pull request may not be mergeable, no matter how much additional effort is applied to it (e.g. out of scope). In these cases, it's best to communicate with the contributor graciously while describing why the pull request was closed, this encourages productive future involvement.

Make sure to:

1. Thank the contributor for their time and effort.
2. Fully explain the reasoning behind the decision to close the pull request.
3. Link to as much supporting documentation as possible.

If you'd like a template to follow:

Thanks ____ for the time you've spent on this pull request.

I'm closing this pull request because _____. To clarify further, _____.

For more details, please see ____ and ____.

Teams

Two GitHub teams are used in the project.

- [Gutenberg Core](#): A team composed of people that are actively involved in the project: attending meetings regularly, participating in triage sessions, performing reviews regularly, working on features and bug fixes and performing plugin and npm releases.
- [Gutenberg](#): A team composed of contributors with at least 2–3 meaningful contributions to the project.

If you meet this criterion of several meaningful contributions having been accepted into the repository and would like to be added to the Gutenberg team, feel free to ask in the [#core-editor Slack channel](#).

Projects

We use [GitHub projects](#) to keep track of details that aren't immediately actionable, but that we want to keep around for future reference.

Some key projects include:

- [Phase 2](#) – Development tasks needed for Phase 2 of Gutenberg.
- [Phase 2 design](#) – Tasks for design in Phase 2. Note: specific projects may have their own boards.
- [Ideas](#) – Project containing tickets that, while closed for the time being, can be revisited in the future.

First published

April 25, 2019

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Repository Management”](#)

[Previous Accessibility Testing](#) [Previous: Accessibility Testing](#)

[Next Folder Structure](#) [Next: Folder Structure](#)

Folder Structure

[↑ Back to top](#)

The following snippet explains how the Gutenberg repository is structured omitting irrelevant or obvious items with further explanations:

```
└── LICENSE
└── README.md
└── SECURITY.md
└── CONTRIBUTING.md
└── CODE_OF_CONDUCT.md
```

```
└── .editorconfig
└── .eslintignore
└── .eslintrc
└── .jshintignore
└── .eslintignore
└── .prettierrc.js
└── .stylelintignore
└── .stylelintrc.json
└── .markdownlintignore
└── .npm_package_json_lintrc.json
└── phpcs.xml.dist
```

Dot files and config files used to configure the various linting tools used in the repository (PHP, JS, styles...).

```
└── .browserslistrc
└── babel.config.js
└── jsconfig.json
└── tsconfig.json
└── tsconfig.base.json
└── webpack.config.js
```

Transpilation and bundling Config files.

```
└── .wp-env.json
```

Config file for the development and testing environment.

Includes WordPress and the Gutenberg plugin.

— composer.lock

— composer.json

Handling of PHP dependencies. Essentially used for development tools.
The production code don't use external PHP dependencies.

— package-lock.json

— package.json

Handling of JavaScript dependencies. Both for development tools and
production dependencies.

The package.json also serves to define common tasks and scripts
used for day to day development.

— changelog.txt

— readme.txt

Readme and Changelog of the Gutenberg plugin hosted on the WordPress
plugin repository.

— gutenberg.php

Entry point of the Gutenberg plugin.

— post-content.php

Demo post content used on the Gutenberg plugin to showcase the editor.

— .github/*

Config of the different GitHub features (issues and PR templates, CI,

— bin/api-docs

Tool/script used to generate the API Docs.

— bin/packages

Set of scripts used to build the WordPress packages.

— bin/plugin

Tool use to perform the Gutenberg plugin release and the npm releases

— docs/tool

Tool used to generate the Block editor handbook's markdown pages.

— docs/*.md

Set of documentation pages composing the [Block editor handbook] (<https://make.wordpress.org/block-editor/handbook/>)

— lib

PHP Source code of the Gutenberg plugin.

— packages

Source code of the WordPress packages.

Packages can be:

- Production JavaScript scripts and styles loaded on WordPress
and the Gutenberg plugin or distributed as npm packages.
- Development tools available on npm.

— packages/{packageName}/package.json

Dependencies of the current package.

— packages/{packageName}/CHANGELOG.md

— packages/{packageName}/README.md

— packages/{packageName}/src/**/*.js

— packages/{packageName}/src/**/*.scss

Source code of a given package.

— packages/{packageName}/src/**/*.test.js

JavaScript unit tests.

— packages/{packageName}/src/**/{ComponentName}/index.js

Entry point of a given component.

— packages/{packageName}/src/**/{ComponentName}/style.scss

Style entry point for a given component.

— packages/{packageName}/src/**/{ComponentName}/stories/*.js

Component Stories to load on the Gutenberg storybook.

— packages/e2e-tests

End-2-end tests of the Gutenberg plugin.
Distributed as a package for potential reuse in Core and other plugins.

— phpunit

Unit tests for the PHP code of the Gutenberg plugin.

— storybook

Config of the [Gutenberg Storybook](<https://wordpress.github.io/gutenberg-storybook>)

— test/integration

Set of WordPress packages integration tests.

— test/native

Configuration for the Gutenberg Mobile unit tests.

— test/unit

Configuration for the Packages unit tests.

— tools/eslint

Configuration files for the ESLint linter.

— tools/webpack

Configuration files for the webpack build.

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Folder Structure”](#)

[Previous Repository Management](#) [Previous: Repository Management](#)

[Next Versions in WordPress](#) [Next: Versions in WordPress](#)

Versions in WordPress

[↑ Back to top](#)

With each major release of WordPress a new version of Gutenberg is included. This has caused confusion over time as people have tried to figure out how to best debug problems and report bugs appropriately. To make this easier we have made this document to serve as a canonical list of the Gutenberg versions integrated into each major WordPress release. Of note, during the beta period of a WordPress release, additional bug fixes from later Gutenberg releases than those noted are added into the WordPress release where it is needed. If you want details about what's in each Gutenberg release outside of the high level items shared as part of major WordPress releases, please review the [release notes shared on Make Core](#).

If anything looks incorrect here, please bring it up in #core-editor in [WordPress.org slack](#).

Gutenberg Versions WordPress Version

16.2-16.7	6.4.2
16.2-16.7	6.4.1
16.2-16.7	6.4
15.2-16.1	6.3.1
15.2-16.1	6.3
14.2-15.1	6.2
13.1-14.1	6.1.1
13.1-14.1	6.1
12.0-13.0	6.0.3
12.0-13.0	6.0.2
12.0-13.0	6.0.1
12.0-13.0	6.0
10.8-11.9	5.9.3
10.8-11.9	5.9.2
10.8-11.9	5.9.1
10.8-11.9	5.9
10.0-10.7	5.8.3
10.0-10.7	5.8.2
10.0-10.7	5.8.1
10.0-10.7	5.8
9.3-9.9	5.7.1
9.3-9.9	5.7

Gutenberg Versions WordPress Version

8.6-9.2	5.6.1
8.6-9.2	5.6
7.6-8.5	5.5.3
7.6-8.5	5.5.2
7.6-8.5	5.5.1
7.6-8.5	5.5
6.6-7.5	5.4.2
6.6-7.5	5.4.0
5.5-6.5	5.3.4
5.5-6.5	5.3.3
5.5-6.5	5.3.2
5.5-6.5	5.3.1
5.5-6.5	5.3.0
4.9-5.4	5.2.7
4.9-5.4	5.2.6
4.9-5.4	5.2.5
4.9-5.4	5.2.4
4.9-5.4	5.2.3
4.9-5.4	5.2.2
4.9-5.4	5.2.1
4.9-5.4	5.2.0
4.8	5.1.6
4.8	5.1.5
4.8	5.1.4
4.8	5.1.3
4.8	5.1.2
4.8	5.1.1
4.8	5.1.0
4.7.1	5.0.10
4.7.1	5.0.9
4.7.1	5.0.8
4.7.1	5.0.7
4.7.1	5.0.6
4.7.1	5.0.5
4.7.1	5.0.4
4.7.1	5.0.3
4.7.0	5.0.2
4.6.1	5.0.1
4.6.1	5.0.0

First published

March 9, 2021

Last updated

January 29, 2024

Edit article

[Improve it on GitHub: Versions in WordPress”](#)

[Previous Folder Structure](#) [Previous: Folder Structure](#)