Heart Internet Presents

# GET STARTED WITH GUTENBERG

Everything you need to know to build and design blocks in WordPress 5.0

Heart Internet

An ebook version of this book is available at:

www.heartinternet.uk/blog/get-started-gutenberg-free-ebook-download/

# Contents

The editor

**Oliver Lindberg**

Oliver Lindberg is an independent editor, content consultant, and founder of the web event series Pixel Pioneers (pixelpioneers.co), based in Bath, England.

Formerly the editor of .net magazine, he's been involved with the web design and development industry for more than a decade and helps businesses across the world create content that connects with their customers.

# Foreword

## by Oliver Lindberg

WordPress is by far the most popular content management system and now powers more than 33% of the web. With version 5.0, it's introduced a significant change: a block-based editor called Gutenberg. The entire editing experience has been rebuilt for media rich pages and posts. Now every piece of content is defined as a block, which can be customised and used wherever needed.

There has been a lot of debate around this release, so we decided to dive right in and help you get to grips with it all. In this practical book, we clear up any confusion, and — if you haven't upgraded to WordPress 5.0 yet — explain how to do so without your content breaking. Then we move onto designing your first Gutenberg block and the tools you can use to jumpstart your designs. Finally, we explore two ways of building blocks — with JavaScript, using a modular approach, or with Advanced Custom Fields.

All our authors are respected WordPress experts, and we even managed to get Mel Choyce, senior product designer at Automattic and WordPress Core Committer, on board to give us an exclusive inside view of how to design a Gutenberg block. At the end you will have a clear idea of how to use blocks and begin building them yourself.

Let's get started!

## The author
## **Joe Casabona**

Joe Casabona is an accredited college course developer and professor. He's also a front-end developer, has a Master's Degree in software engineering, and hosts multiple podcasts.

Joe started freelancing in 2002, and has been a teacher at the college level for over 10 years. His passion in both areas has driven him to build Creator Courses (creatorcourses.com), a school for those who want to create online businesses.

As a big proponent of learning by doing, he loves creating focused, task-driven courses to help students build something. When he's not teaching, he's interviewing people for his podcast, How I Built It (howibuilt.it).

# Your Upgrade Path to WordPress 5.0: Options and Recommendations

by Joe Casabona

With the rollout of WordPress 5.0, users get to experience a brand new editor. An editor that promises to be more flexible than before; one that allows users to create rich layouts without the need for a lot of shortcodes or plugins. But there also seemed to be a lot of build-up surrounding the launch of WordPress 5.0.

Part of it is because the new editor requires new user education. It's a different experience than what many users are used to, so there's bound to be some confusion.

There's also the possibility that content, themes, and plugins could break with the new editor and the wrong conditions. Couple that with the short timeline from beta to official release, and many people using WordPress were understandably nervous — perhaps they've even held out on upgrading. If this is the case for you, fret not! In this article, we're going to look at our options for upgrading to WordPress 5.0, and a recommended path.

## The current WordPress landscape for updates

First, we should clear up some confusion around WordPress automatic updates: They are only for minor releases (4.9.x, 5.0.x). Any major releases (4.x, 5.x) require a manual update — either you'll have to do it yourself, or your hosting company will run a script to do it for you (more on that later). Heart Internet gives you the opportunity to do it yourself with your existing site, but new sites will be on the latest version.

That means if you (or your host) didn't upgrade to 5.0, your site's likely still on 4.9.9. It's also worth noting the security updates and compatibility changes are back-ported to older versions, so 4.9 should be secure for the foreseeable future (though I'd definitely keep an eye on WordPress update news to make sure that is the case).

### WordPress updates are becoming more frequent

Before 2018, WordPress was basically updated with a major release two to three times per year, on a predictable schedule. In December 2017, it was announced that major releases would come out "as major features are completed." That's why we saw a year before 5.0 came out — the new block editor wasn't ready before.

It also means that we'll probably see a lot more major releases in 2019. Version 5.1 came out in February, and 5.2 in May 2019, while 5.3 is in active development. That means we'll have more control over exactly when we update WordPress.

### How hosting companies are handling WordPress 5.0

Depending on your hosting company, the WordPress 5.0 roll out was a different experience. There are a few things hosting companies were known to do:

1. Block the 5.0 update completely. I know of at least one that is still doing this, taking advantage of the back porting to 4.9.x, and testing 5.0 to make sure it really works with their systems.
2. Automatically update and immediately enable the classic editor.
3. Move forward with the update, with fair warning to their customers.
4. Let their users decide (which is Heart Internet's choice and also the default WordPress functionality).

In each instance, the hosting company did what they felt was best for their customers, based on their offerings and systems. But assuming you're not wholesale blocked from upgrading to 5.0, we can take a closer look at each of these options to see what's best for you.

## Your upgrade options

Knowing that the new block editor in 5.0 could cause some compatibility issues, the WordPress Core team also created a plugin called Classic Editor **[1]**, which would turn off the block editor, or give you the option to switch between the two. There's also a plugin called Gutenberg Ramp **[2]**, by Automattic (the company behind WordPress.com), which allows you to turn the block editor on per post or post type. This gives you several options for upgrading.

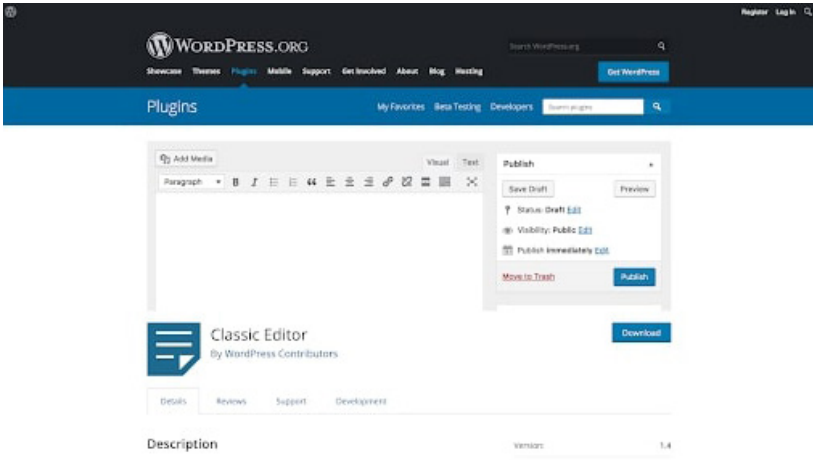
## Don't update (not recommended)

You can do nothing. As stated earlier, the most critical updates will be back ported to WordPress 4.9 for the foreseeable future. However, the ecosystem still changes. Plugins and themes will update, and servers will only support a minimum PHP version for so long. Not to mention that there are other benefits to upgrading to 5.0, especially as the next set of major releases comes out.

## Update and install the Classic Editor

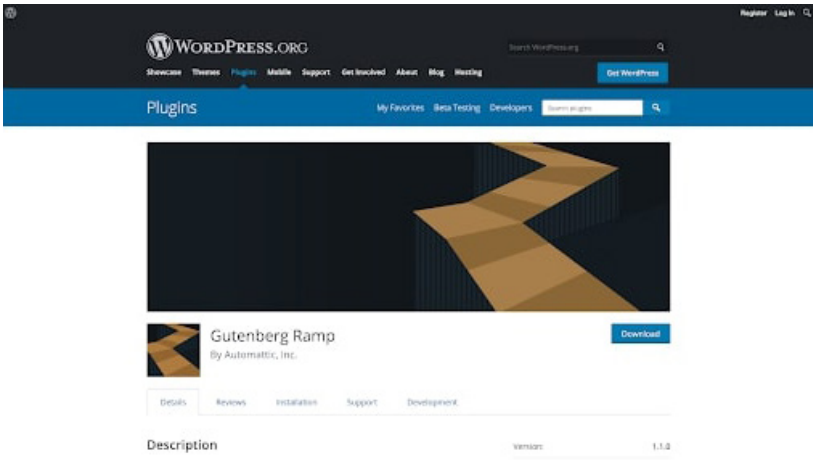
You can also update to WordPress 5.0 and immediately install the Classic Editor. An important thing to note here is upgrading to 5.0 does nothing except enable the new editor for new content. Your current content will not change until you open it in the new editor, then convert it.

That means if you immediately install the Classic Editor and enable it, you have effectively disabled the only major change from 4.9 to 5.0. This is an excellent way to upgrade to 5.0 without breaking anything and to give yourself time to test. A good stopgap in general.




## Slowly roll out the new editor with Gutenberg Ramp

Similarly, if there's content you know will work with the block editor, or want to use the block editor for new content, you can use Gutenberg Ramp. This plugin will give you more flexibility than the Classic Editor, and it will allow you to embrace the block editor for new content, while maintaining the compatibility for older content.




## Just hit upgrade (not recommended)

Finally, you can just hit upgrade. This, like doing nothing, is not recommended. As we'll get to in the next section, the new block editor stands to break content and functionality, and just hitting upgrade might cause irreversible changes to your site (so at the very least, make a backup).

## Definitely test, no matter what you do

No matter what you decide to do, it's important to test your website to see what you can expect from the upgrade. Here's how to do that.

# Testing for WordPress 5.0

When it comes to testing your WordPress website for 5.0 and the new editor, there are a few areas you'll need to look at: you content, your plugins, and your themes.

## Testing your content

First and foremost comes your content. After all, the block editor will affect it the most, right? Well, that really depends. The thing with your content, as we saw earlier, is the new block editor will only affect it if you open that content in the new editor. And most likely, it won't change until you convert it to blocks.

That said, if you plan on using the new block editor full time, you will need to create a list of content to test. It should include:

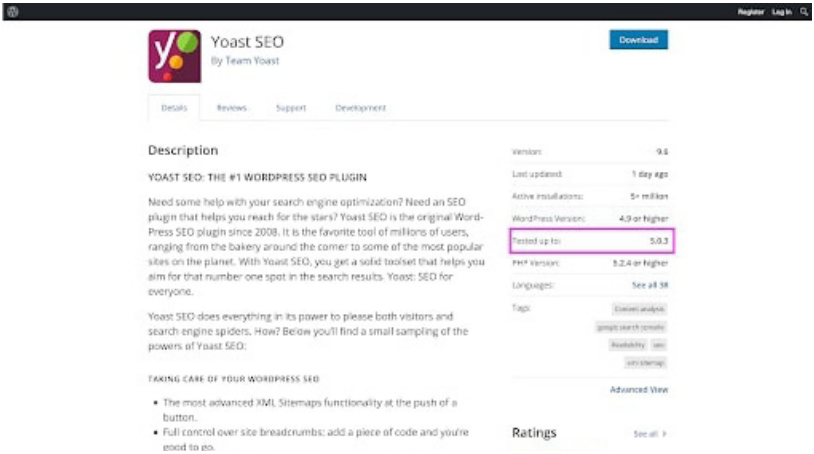
- Posts
- Pages
- Custom Post Types
- Content based on specific templates
- Content created with a page builder
- Content that is plugin- or theme-dependent

You'll also need to consider how much custom HTML, CSS, or PHP you have in the editor itself. Have you relied heavily on the "Text" view of the Classic Editor? You may run into several issues there. At the time of writing, the new block editor doesn't do a great job (understandably) of converting custom HTML.

We recommend taking an inventory of all the types of content you have and making a list to spot check your site. Open the content on the front-end, then in the block editor, then convert and upgrade, and check the front-end again (all on a staging site, of course).

## Testing your plugins

Once you've inventoried your content, take stock of the plugins and determine which need testing. Be mindful of the ones that rely on the editor in some way, shape or form. That includes plugins that add meta boxes — you'll want to make sure everything plays nicely with the new editor:




Check the "Tested up to" area to make sure your plugins are compatible with 5.0.

And now that 5.0 has been out for a while, plugins have had time to test and update their compatibility with the latest version of WordPress, so check that for each plugin as well.

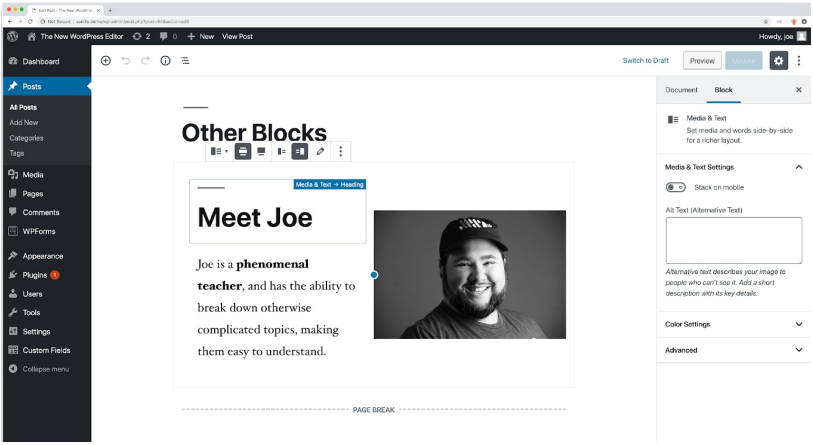
Also note that plugins that don't touch the editor at all likely won't be affected by 5.0. Even the ones that only do things like add widgets or shortcodes are already accounted for. That said, they're still worth testing.

## Testing your theme

Your theme is where you'll see the highest visible risk in upgrading to WordPress 5.0. The new editor adds new types of content, as well as new classes — alignwide and alignfull — that your theme may not support right away. And the editor itself only adds minimum styles to make them work properly on the front-end.

If you're using a theme from the Themes Directory, or a premium theme, make sure that it's updated to support the new blocks in 5.0. If you're using a custom-made theme, you'll need to do a few things:

- Add CSS to support new blocks
- Add some generic classes that can be added to blocks
- Add theme support **[3]**
- Test every template




The "Media/Text" block in action.

A few blocks worth testing explicitly are:

- The Cover Image
- The Gallery
- Columns
- Pull Quote / Block Quote
- Media / Text
- File

# How to upgrade to WordPress 5.0

With our options available, and a simple test plan created, here's how we can upgrade to WordPress 5.0.

## 1) Inventory your website

As suggested above, you should inventory your entire site. Make a list of all the types of content you have on it, make a list of your plugins, and check your theme.

With your plugins, check to see what has been updated to be compatible with 5.0. Rank them to determine which are most-likely and least-likely to break.

After you inventory everything, make a list of things you commonly do in WordPress. What's your normal content creation flow? Do you copy and paste from Word or Google Docs? Do you rely on outside APIs? Write them down for you to test later.

## 2) Create a staging site

You don't want to do this on your live site. Create a backup of the live site, and make a staging site for you to work from. Most hosts support doing this in some way or another.

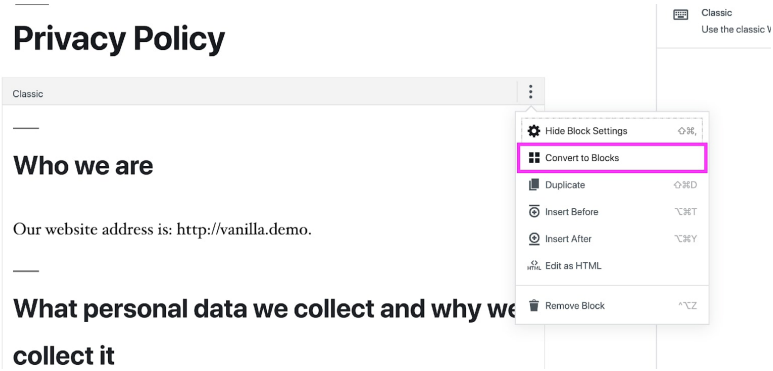
## 3) Upgrade everything

On your staging site, head over to the Updates page and hit the upgrade button for WordPress 5.0.

Then do the same for your themes and plugins.

## 4) Run through your inventory

Once you're upgraded, before doing anything else, check your site on the front-end. Click through all the pages, review some posts, and check any other content you laid out in your inventory. This should be the control: you're making sure nothing broke just by virtue of upgrading.




You can convert the Classic Editor to blocks using the "More Options" menu.

Then take that same inventory and check all of the content and templates. Open your content in the new editor and convert the content to blocks. Make changes to the content and view the content on the front-end.

Do the same thing with your plugins. Check to see how they work with the new editor, and if anything is broken.

Finally, make a "Kitchen Sink" page with all of the new blocks. Make sure to use the colour pickers, and create blocks using the new alignment classes as well. Open that page on the front-end to see how your theme works with the new editor.

## Evaluation time

Once you've run through your tests, it's time to evaluate. If everything went smoothly and nothing broke, congratulations! You can upgrade the live site to WordPress 5.0.

If something broke in a specific area, or a specific type of content, but everything else worked fine, you should consider Gutenberg Ramp. That way you can upgrade most of the site to the new editor, while working to fix the problem areas.

If lots of stuff broke, install the Classic Editor and make a list of everything you need to fix, then come up with an upgrade plan.

### Your final steps

If you are stuck on the Classic Editor for now, be aware that you can't stay there forever (though it will be supported through 2021). You should make your upgrade plan to fix what needs fixing. You can also report issues to the WordPress Core team.

## You did it!

Congratulations! You have a clear plan for upgrading your website to WordPress 5.0, even if you're not using the new block editor just yet. Upgrading now will make life much easier for you in the future, as the Gutenberg project continues to permeate different areas of WordPress.

### Resources

[1] Classic Editor plugin
https://wordpress.org/plugins/classic-editor/

[2] Gutenberg Ramp plugin
https://en-gb.wordpress.org/plugins/gutenberg-ramp/

[3] Gutenberg Handbook
https://wordpress.org/gutenberg/handbook/designers-developers/developers/themes/theme-support/

The author
**Mel Choyce**

Mel Choyce is a senior product designer at Automattic based in Boston, Massachusetts. Mel is a WordPress Core Committer and former Release Lead, and a regular core contributor. She speaks frequently at WordCamps on design, typography, and user experience.

When Mel isn't designing products at Automattic, she enjoys cold brew coffee, craft beer, and rocking out in her band. Say hi to her on Twitter at @melchoyce, and visit her site at choycedesign.com.

# Designing Your First Gutenberg Block

by Mel Choyce

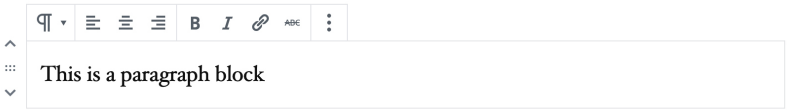
Gutenberg is the future of WordPress.

For most of its existence, WordPress has operated on a "document" model of writing. You have an empty container that you fill with text from top to bottom. There's a docked formatting toolbar on top of that container. This is an interface familiar to anyone who's ever worked in Microsoft Word or Google Docs.

The web has evolved beyond displaying digital documents, but WordPress has only just started to catch up. The introduction of Gutenberg blocks takes us beyond the document model of web authorship, towards an approach that looks less like writing and more like *building*.

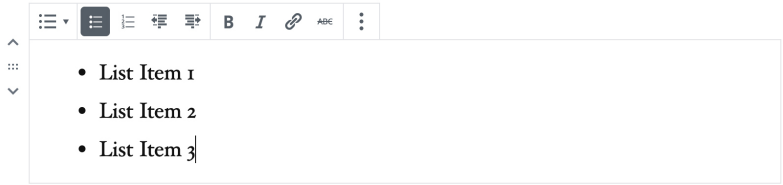
Gutenberg blocks are like Lego — each a self-contained unit that can be combined to make some pretty fantastic outcomes. Like Lego, the possibilities are only constrained by your imagination.

You don't need to be a designer to create a block. In fact, you, and most other block builders, are like a developer. That's fine. You don't need to be a designer to make some useful, intuitive Gutenberg blocks. You just need to think a little bit like one. With some critical thinking and familiarity with *core* Gutenberg components and patterns, anyone can create a great block. At its simplest, a block can be a paragraph.




The Paragraph block has a prompt that encourages you to write. You can add inline formatting, like bold or italic text, add links, and change the alignment. If you want to turn your paragraph into a heading instead, you can do that, too. This is very similar to how WordPress has always operated.

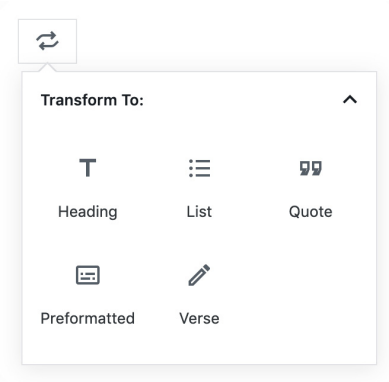
Let's take a look at another simple block, the List.




In a document-driven editing model, you would start a list by applying list formatting, via a toolbar, to existing text, or you'd go to the formatting toolbar and click 'List' to create a new bullet point.

This is similar in Gutenberg. You can transform a Paragraph block into a List block, or add a new List block directly to your page.

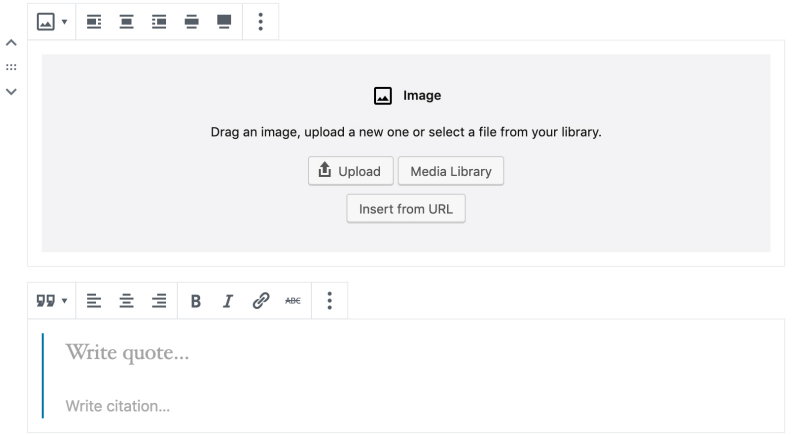
Some blocks require more than a simple text prompt. More complex blocks require more advanced setup states.




For example, the Image block creates a placeholder with buttons for uploading images, selecting existing images from your media library, and for inserting an image from a URL.

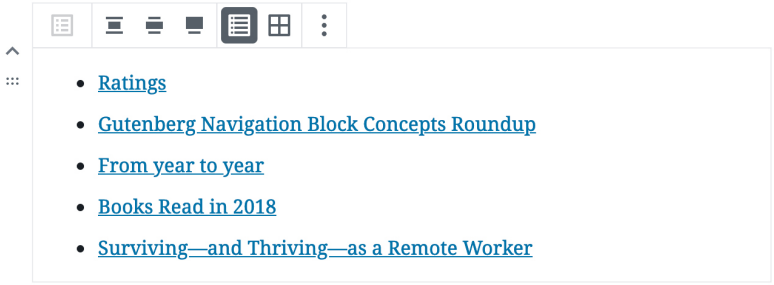
Similarly, the Quote block has two text inputs: the quote, in larger text, and the citation, in smaller text.

Some blocks are much more complex and need a setup state.



The Table block requires you to select the number of columns and rows before it can render the correct placeholders. A Google Maps block could require an API key to function.

Or a block might be drawing dynamic content from a different part of your site — like a post, or a custom post type. The Latest Posts block, by default, displays the title of your most recent five posts, from newest to oldest.




No additional input is needed for the block to work, but additional configuration options are available via the block toolbar and sidebar, which we'll explore next.

# The anatomy of a block

Gutenberg blocks are comprised of a few different elements:

- The icon
- The content area
- The toolbar
- The sidebar

## Block icons

Each block has an icon used to identify it across the various block interfaces, such as the block library, the toolbar, the sidebar, and occasionally a setup state.


Icons are generally displayed at 24x24 pixels, but should be scalable. Use SVGs. You can pick from an existing Dashicon **[1],** design your own custom icon, or use an icon from a third-party, GPL-compatible library. Many of the new icons in Gutenberg are based on Material Design's outlined icons **[2].**

When selecting or designing a block icon, think about your block's primary purpose. Some blocks are obvious — an Image block should use an image icon, a List block should use a list icon, and a Quote block should use a quotation mark icon.

Some blocks aren't so obvious. What icon should a Featured Content block use? How about a Carousel block? CTA? Pricing Table? You might need to get creative. Look through icon libraries, and see if they have anything that fits your block purpose. Search for your block name in something like IconFinder (iconfinder.com) and see what imagery it pulls up. See if a website building service like Squarespace or Weebly has a comparable block, and what icon it's using.

When all else fails, go for the most generic representation you can find — maybe a shape, or something nondescript — and be happy your block will usually show up alongside its name.

## The content area

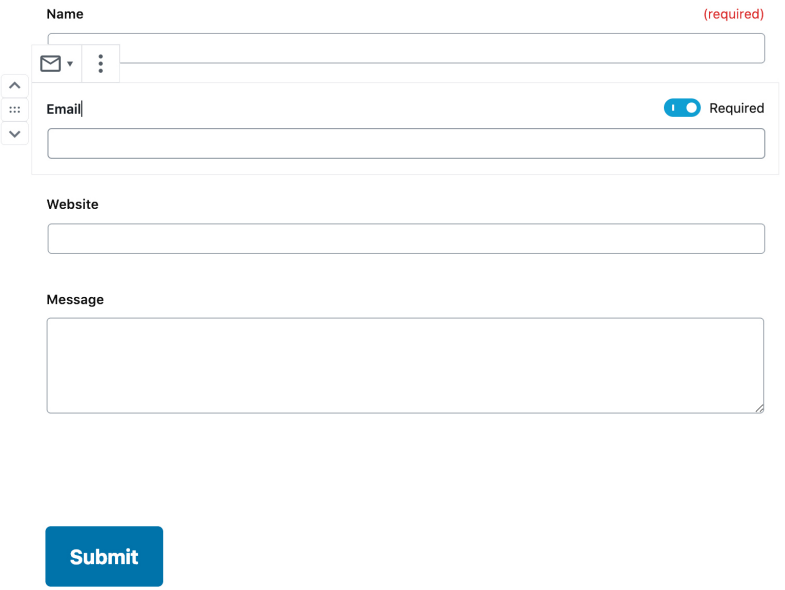
> "The primary interface for a block is the content area of the block."

— The Gutenberg Block Design documentation **[3]**

In an ideal world, the only thing you'd ever have to think about when adding content to a block is your content itself. What does this mean in practice?

**If a block needs specific information to render, ask for that information in a setup state.** Need an API key? Ask for that in your setup state. Need posts from a specific taxonomy to display? Show a list of existing taxonomies in your setup state.

**The most important settings should display inline with your block.** Take, for example, the Jetpack Contact Form block.



The “required” setting, which exists per-field, is displayed alongside the input label. When you select the input block, you see a toggle to turn “required” on or off. When the input field is not selected, you see a live preview of the input.

Your live preview state, when your block isn’t selected, should always mimic the front-end as closely as possible.

**Choose smart defaults.** Don’t make your users or customers have to think too hard about setting up a block. Do some initial research, whether that’s a couple of interviews with existing customers, or finding existing, comparable research from a reputable source like the Nielsen Norman Group (nngroup.com), and determine which settings most of your customers change. Make these changes the default options.

This suggestion trumps the previous tips. If you *can* provide smart defaults, do.

## The toolbar

> "The block toolbar is a secondary place for required options & controls."

— The Gutenberg Block Design documentation **[4]**

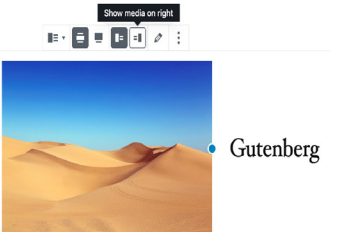
By default, the toolbar is displayed above the block, when it's selected.

You can also show the toolbar at the top of your editor if you have the Top Toolbar setting enabled.

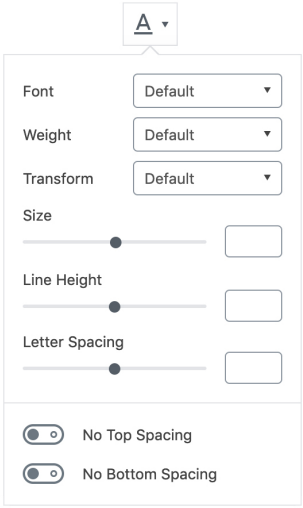

Regardless of your block settings, all toolbars show the block icon, and a "more" menu with options to duplicate, move, delete the block, etc. Everything else is optional.

The toolbar is a great place to include inline formatting options, like **bold**, *italic*, ~~strikethrough~~, and links.

The toolbar is also the right place to include block-level alignment and layout settings for your block. In the Media & Text block, for example, there's a Toolbar setting to display the media on the left, or the right.



The CoBlocks plugin (coblocks.com) adds a custom font option to your blocks, which fits well into the toolbar:



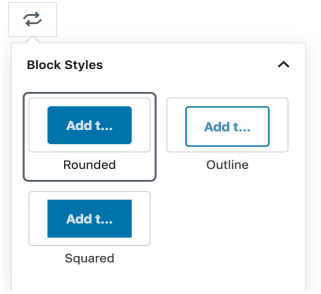

One caveat: the toolbar is icon-based, so only add settings you can reasonably represent using an icon, or an icon group. There will be tooltips on desktop computers, but any touch-based device won't necessarily have labelling you can rely on.

The toolbar also houses two additional options: styles, and block transforms.

Block **styles** are alternate variations on a block **which only rely on CSS.**

Take, for example, the Button block, which comes with three core styles: Default (rounded corners), Outline, and Squared. When these styles are available, they're accessible via a dropdown next to your block icon:

Choose a smart default, but feel free to include different styles for your custom blocks, or extend existing core blocks in interesting and unique ways.



You can also **transform** similar blocks. A paragraph can transform into a Heading, Quote, List, Preformatted, or Verse block. An image can transform into a Gallery, Media & Text, Cover, or File block. Is your block similar to another core block, or another block you offer? Add a transform option. Similar to styles, transforms exist in the dropdown next to your block icon.

Make sure that if you do include a transform, the existing content converts smoothly to the new block type. You don't want your users or customers to lose any data.

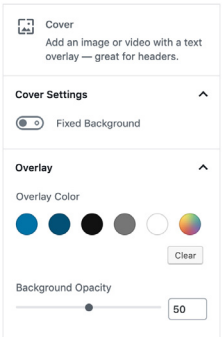
## The sidebar

> "The block sidebar should only be used for advanced, tertiary controls."

— The Gutenberg Block Design documentation **[5]**

The sidebar, as you might guess, appears on the side of your editor. You can toggle it on or off, which means you can never rely on the sidebar being open. In fact, the sidebar is hidden by default on smaller screens. This makes it a good place for advanced or non-essential settings, but a bad place for any required or important settings.

Settings within the sidebar should be grouped into sections based on similarity. The Paragraph block groups together text settings like font size and a drop cap toggle into a section. The Cover block




groups positioning settings like 'fixed background' and 'focal point' into one section, and overlay settings like 'color' and 'opacity' into another section.

You might be tempted to add a ton of settings to your block, since they can all fit into the sidebar. However, I'd urge you to think very carefully about what settings you add to your blocks.

Firstly, once you include a setting, you're going to have to support that setting — perhaps indefinitely. If you include a setting on a whim and then suddenly end up with a constant stream of support questions about that particular setting, you might regret including it in the first place. At the very least, you should redesign the setting to make it easier to understand.

Secondly, the more settings your block includes, the more complicated it becomes. Think about your audience — who do you expect to use your blocks? If you're aiming for site builders and freelancers, then adding a bunch of customisation options might be appropriate. However, if your block is aimed at casual users or hobbyists, you should consider paring down your settings to the minimum needed for your block to be useful.

Speaking of more advanced settings, every block comes with an "Advanced" section that houses an "Additional CSS Class" option. If a user adds a class name to this, that class will be applied to your block. If you have very advanced settings, especially developer settings that can't easily be explained to your average user, put these in your Advanced section.

## Designing your block

Now that we know all of the elements that make up a block, let's put it all together.

In December 2018, I designed a Restaurant Menu block **[6]** .I've wanted a better way of making restaurant menus since working on WordPress.com's restaurant menu custom post type a couple years ago. The custom post type method of creating a restaurant menu felt clunky and tedious. Blocks provide a perfect opportunity to redesign this process. Gutenberg blocks put the interface within the context you'll be displaying and previewing the menu in, rather than abstracting the interface out into a separate, unrelated screen. Text placeholders allow for an easy "fill-in-the-blanks" method for content creation. It seemed like a natural fit.

### Scoping the block

Going into this project, I assumed the vast majority of restaurant menus are pretty similar — but I wanted to validate those assumptions before I started to put pixels down onto my screen. I looked up the menus for over a dozen restaurants I like and jotted down what kind of information they included. Menus were divided up into different categories, such as Appetizers, Entrees, Desserts, etc., some of which had descriptions. Almost every menu included at least a title and price for every individual menu item. Many also included descriptions. There were a scattering of other features — some menus included photos, some included how spicy a dish was, and others marked whether a dish was vegetarian or gluten free. Some menus even allowed filtering, either by category or by diet.

This is potentially a lot of features. Let's pare it down to a minimum viable block:

- Menus should be divided up into sections, each with a heading.
- Each menu item should contain a title, price, and optional description.

That's all we really need for a restaurant menu. Each section could be a parent block that includes a heading, with menu items as children.

Why a block per section, instead of one block for the entire menu? By breaking out the block into a menu section, your menu can be easier to rearrange, style, and scale.

A few additional options would make the block more useful for many restaurants:

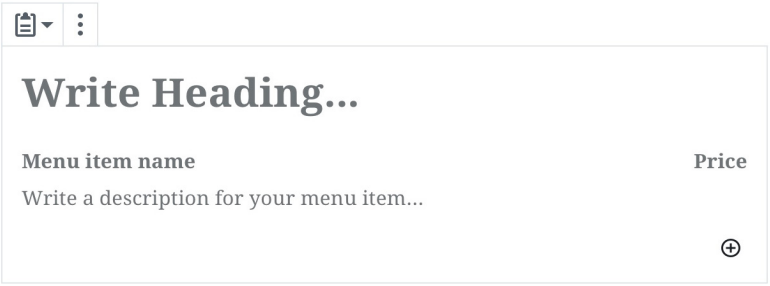
- The ability to intersperse additional headings, paragraphs, and images into a menu. This would cover a lot of the miscellaneous use cases I saw, such as section descriptions and the occasional image highlighting a specific dish.
- Column support. Many of the menus I saw broke sections up into multiple columns to better use up space on the page.
- An optional field to mark any allergens in a dish, or say whether a dish is compatible with a particular diet.
- Additional block styles. While the default styles would show title and description on the left, with price on the right, we could add another style in which title, description, and price are centred on the page. This was a popular style amongst the restaurant menus I researched.

## Refining the settings

Now that we've scoped our Restaurant Menu block, let's figure out where each piece of the block should live.

### Setup state

The setup state should include placeholder text for the most important content in the menu: the section heading, and one empty menu item that includes a name, price, and description. This establishes all your baseline patterns and starts teaching people how to use your block.

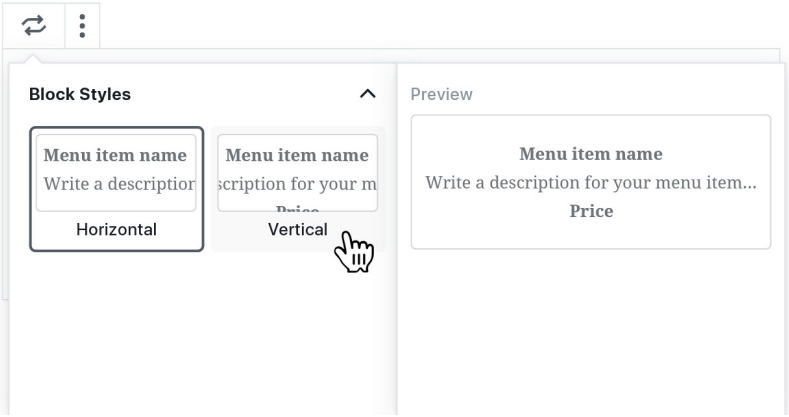



For convenience, the block should default to adding a new menu item, so all people need to do to get another item is press "enter" at the end of the description field. People can still press the "down" key to get outside of the parent block.

### Toolbar

Because we have a parent block and a child block, each will have its own toolbar.

Looking through our scope, the only setting that seems to belong inside the parent toolbar, the "Restaurant Menu," is our alternate menu style:



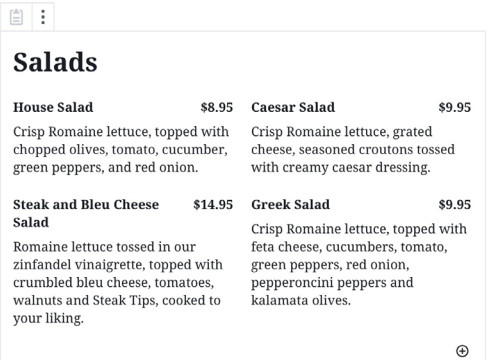

I thought about adding support for wide- and full-width layouts to the block, but the majority of menus I saw were constrained to the content width. If a bunch of people start asking for this feature later, it's easy enough to add support.

Our child block, "Menu Items," consists of our title, price, and description. These fields could benefit from some very basic text formatting. Bold and italic seemed reasonable, so I decided to include those. Strikethrough and link, which are available in the paragraph block, feel unnecessary in this context, so I excluded them. But once again — should they prove to be desired, we could always add support in a future release!

### Sidebar

What's left to include in our block? There are only two settings we haven't talked about yet: columns, and our allergen toggle.

I found plenty of menus that used a one-column layout in my research, but just as many used two or more columns. It seemed like a setting worth supporting, and Gutenberg already comes with column support available.

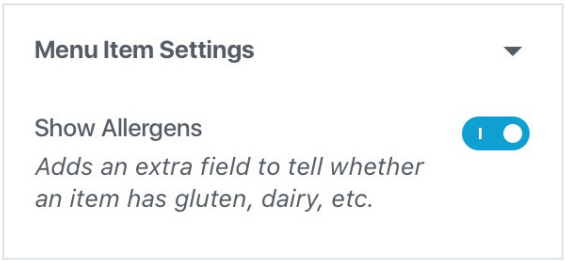



We probably don't want to allow people to add infinite columns to the menu block, so it would make sense to limit the block to a maximum of maybe four columns.

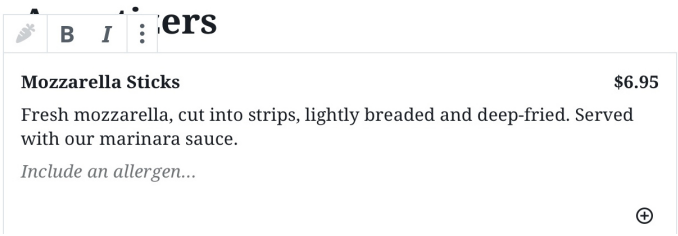
Let's talk about the allergen toggle.

(I don't think this is necessarily the best label, but I couldn't think of a better word at the time. I didn't want to use "diet" because it means different things to different people.)

Toggling "allergens" on adds a new field underneath the item description, so you can add some extra details like "gluten-free" or "vegan":




Including a description within the sidebar setting makes it clearer what the setting does, especially when it has a more ambiguous label.




When you toggle the setting on, a new field appears below the menu item description field.

Because not every menu item will need to include an allergen, this is a setting you'd turn on for the individual menu items that need it.

**The block icon**

Let's not forget about the block icon! This block will need two: one for the parent block (Restaurant Menu), and one for the custom child block (Menu Item). I looked through some icon libraries, looking for inspiration. I found a clipboard that made me think of some menus I've received in restaurants, so I chose that for my parent block. I remembered Dashicons has a carrot icon **[7],** which seemed like a good way to illustrate food.

(If I was redesigning this now, I'd probably go back and use Dashicon's clipboard icon **[8],** which I missed when I first designed the block.)

Now that we've scoped out this block, we can finally build it! To learn more about how to build a block, check out the chapters on building blocks with JavaScript and doing the same with Advanced Custom Fields.

## Usability testing and iteration

Okay, you've built your shiny new block. Now what?

Before launching your block to the world, consider doing some usability testing. If you're part of a larger team with more resources to spend on research and testing, you can pay someone to recruit and run usability tests for you — or even use your in-house team of UX experts! If you're a lone plugin or theme developer, however, your testing doesn't have to be super comprehensive — you can keep it small and simple.

There are better guides for telling you how to conduct usability testing than I can in this article, such as Rocket Surgery Made Easy **[9]** by Steve Krug. You can read a sample chapter *online* **[10].** If you want something more condensed, this article from A List Apart **[11]** is a great introduction.

Once you've run at least three tests, sit down and sift through your recordings or your notes. Find the biggest pain points in your block, and then iterate on them. If you have the time and budget, test again with some new volunteers and see if your changes improved the experience of using your block.

You can iterate ad nauseum, but at some point, you need to ship. Once you have a big enough user base, you'll probably start hearing feedback, suggestions (and of course, bugs) directly from your customers. Good luck, and happy block building!

## Resources

[1] Dashicon
https://developer.wordpress.org/resource/dashicons/#facebook

[2] Material Design's outlined icons
https://material.io/tools/icons/?style=outline

[3] Gutenberg Handbook
https://wordpress.org/gutenberg/handbook/designers-developers/designers/block-design/#the-primary-interface-for-a-block-is-the-content-area-of-the-block

[4] Gutenberg Handbook
https://wordpress.org/gutenberg/handbook/designers-developers/designers/block-design/#the-block-toolbar-is-a-secondary-place-for-required-options-controls

[5] Gutenberg Handbook
https://wordpress.org/gutenberg/handbook/designers-developers/designers/block-design/#the-block-sidebar-should-only-be-used-for-advanced-tertiary-controls

[6] Restaurant Menu block
https://choycedesign.com/2018/12/14/creating-a-gutenberg-restaurant-menu-block/

[7] Carrot icon
https://developer.wordpress.org/resource/dashicons/#carrot

[8] Clipboard icon
https://developer.wordpress.org/resource/dashicons/#clipboard

[9] Rocket Surgery Made Easy book
http://sensible.com/rsme.html

[10] Sample chapter
http://sensible.com/downloads/rsme-samplechapter.pdf

[11] Usability testing demystified
https://alistapart.com/article/usability-testing-demystified

The author

**Sarah James**

Sarah is a Lead User Experience Designer at global digital agency 10up (10up.com), a digital agency with award-winning strategy, design, and engineering serving Fortune 500 companies, influential content publishers, important startups, and non-profits around the world. With nearly a decade of experience in the UX field, Sarah brings her expertise in user-centred design, information architecture, and digital strategy to every design problem. Sarah is also passionate about design research and open source initiatives and has worked in the WordPress community since 2015.

When not focusing on design interactions, you can find Sarah brewing beer, bowling, or being a Dungeon Master for her Dungeons & Dragons group.

# Jumpstart your Gutenberg Designs with SketchPress

by Sarah James

Strong design supports good development. Here at 10up (10up.com), our work is led and continually informed by user experience and interactive design. The same goes for WordPress core **[1].** Even if you are a team of one, there is value in thinking through the user experience and spending time on the design before diving into the development.

Historically, this has required unique skill sets and time spent on repetitive design tasks, which has been out of reach for software engineers and others not specifically trained in user experience and design. This tutorial will show you that with a little time and the help of the SketchPress **[2]** libraries, you can create compelling designs for your WordPress and Gutenberg projects.

Getting buy-in for design work can be a challenge, especially when design changes impact the authoring experience. However, when working in any CMS, this is a core task for appropriately designing content. SketchPress, a library of WordPress admin interfaces, symbols, and icons from 10up, extends Sketch (sketchapp.com), the digital design app for macOS, to enable the rapid creation of wireframes.

SketchPress frees up designers' time to focus on big-picture problem solving and user experience challenges, rather than the repetitive minutiae of button treatments and existing page layouts. This workflow is particularly powerful for creating custom blocks in the new Gutenberg world with the release of WordPress 5.0.

Recently, SketchPress was recognised as the preferred admin wireframing resource by Make WordPress Design **[3]** — the official core WordPress design team — and is now listed on their official Trello project board. This endorsement validates the utility of the project while inspiring us to continue iterating.

## Getting started with SketchPress and libraries

SketchPress includes over 60 admin symbols, nearly 100 icons, 30 common text styles, and the WordPress core colour palette **[4].** Symbols provide an easy way to iterate and reuse elements. If they aren't your preferred workflow, there is also a SketchPress file in GitHub **[5]** with over 20 page shells and typical pattern layouts for posts.

You need Sketch to use these files. If you are unfamiliar with it, we recommend their Getting Started guide **[6].** Before you begin, go to the GitHub resource for SketchPress. There are several files here, including templates for the classic editor and the recent WordPress release (all labelled "WP5.0").

The file used for this tutorial is "WP5.0_Library". Download it, and open Sketch to add it to your library. Go to system preferences, then libraries. At the bottom of the pop-up window is a button to "Add Library..." Find the WP5.0_Library file you just downloaded and add it. Now all of the symbols are ready to use and available for any of your projects. For more information about libraries, leverage Sketch's great resources **[7].**

## Tutorial: Create a custom block with SketchPress

Let's run through an example — creating a custom block. Now that you have the library installed, you can begin mocking up your designs.

Take, for example, this simple visual design of a featured story. This could represent a wireframe, a full visual design comp, or something you've sketched on a napkin.

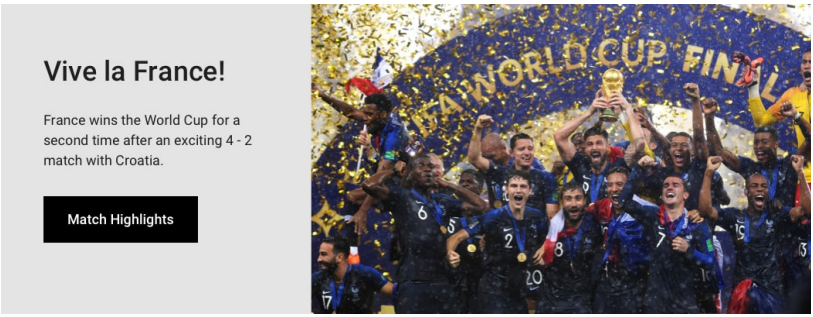
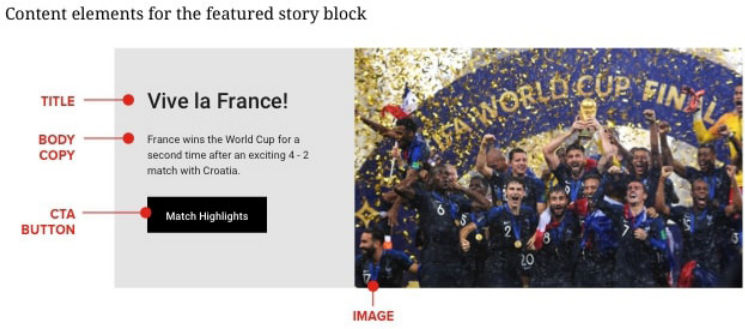



Image source - Wiki Commons

First, document all of the component elements on the block, like in this picture:

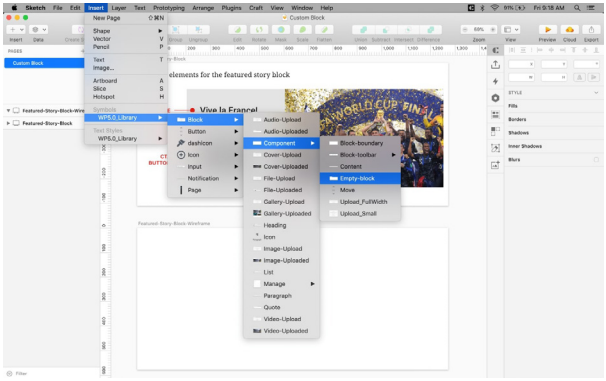



The component diagram serves as a checklist for each element to create in the customised block. Here is a list of what we need to include in our block:

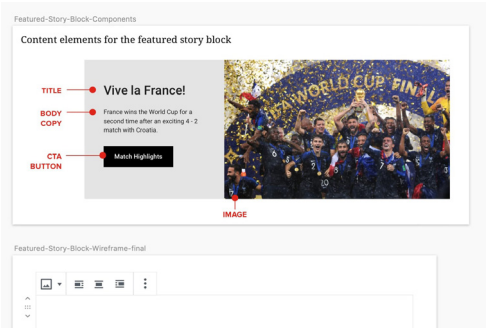
- A title
- Body copy area
- Call-to-action button (with hyperlink address)
- Image

Start with a new Sketch document. If you want to familiarise yourself with your newly installed library, you can go to **Insert > WP5.0_ Library.** Here you will see all the individual elements available to create wireframes.

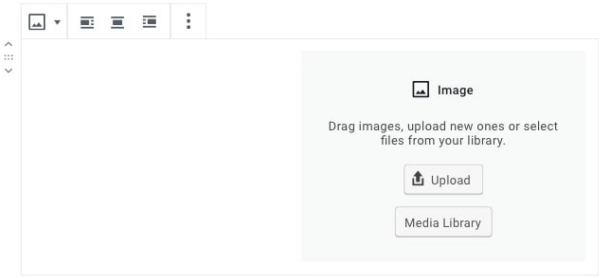
The first thing we want to add to the page is a blank block. As an alternative, you can also reuse a current core block rather than create a new one. In this tutorial we're going to start from scratch. To insert this particular symbol into your document, go to **Insert > WP5.0_Library.** All the symbols for this library are here. What we are looking for is **Insert > WP5.0_Library > Block > Component > Empty Block.**



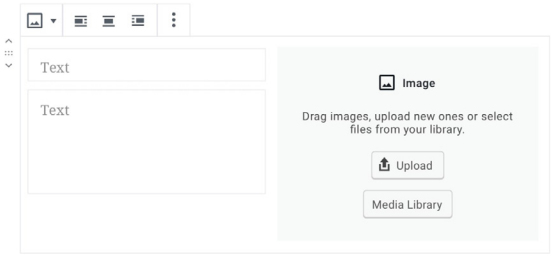

This gives us a nice canvas to start from. In the visual below, you can see both the featured block as a reference and the block we just inserted into our Sketch canvas.



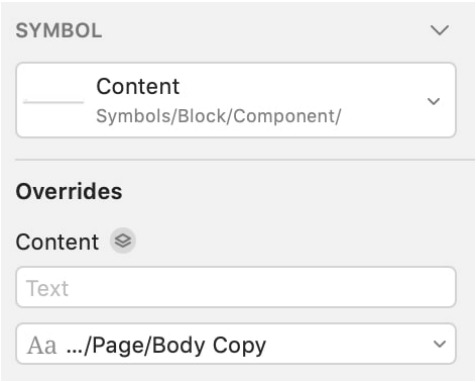
Next, let's add an image upload box. Since we want to keep the backend experience consistent, we can repurpose what is used in the insert image block. Go to **Insert > WP5.0_Library > Block > Component > Upload_Small** to insert the image block into our empty custom block.




Now let's add a title and body copy placeholder. **Insert > WP5.0_ Library > Block > Component > Content.** This adds some blank text boxes into our canvas. We will create two — one for the title and one for the body copy.

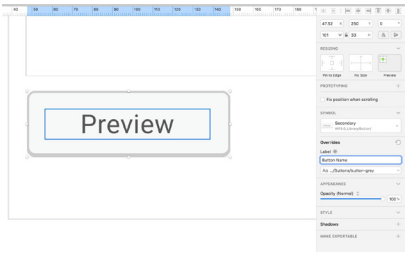


Notice that each of the blocks say “text.” We can customise this by overriding it in the inspector panel (this is the content located on the right side of your Sketch interface). There is an area that says “overrides.” The text input box is the default language for this symbol. Select the input field and add your own text. This will then change the symbol’s text for that particular instance (and not globally).

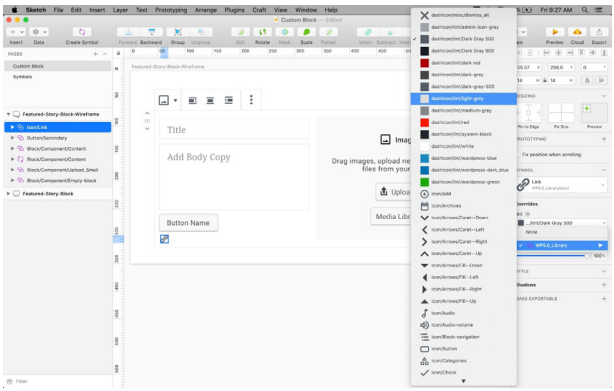



The custom block now has an image placeholder and the title and body copy. All that is left is the button and a URL text section attached to it. As you may have already guessed, we have a symbol for buttons. For this demo, we are going to use the secondary button style in WordPress. That’s **Insert > WP5.0_Library > Buttons > Secondary Button.**

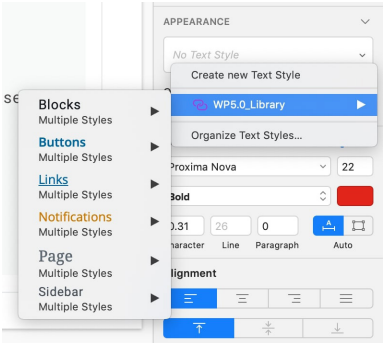
Once again use the override field to change the button label from “Preview” to what you want it to be (in this example, it’s changed to “Button Name”).



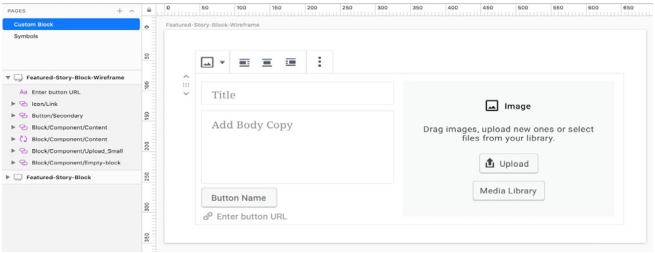

Finally, we are going to add the field for the link text. SketchPress comes with a variety of common icons already added to the library. In this example, we are going to use **Insert > WP5.0_Library > Icon > Link.** When you select the link icon, you see that the inspector panel on the page has options for tints. SketchPress has the WordPress core colour palette that you can select from to easily change the tint of any icon. In this example, it's been changed to be a lighter grey.



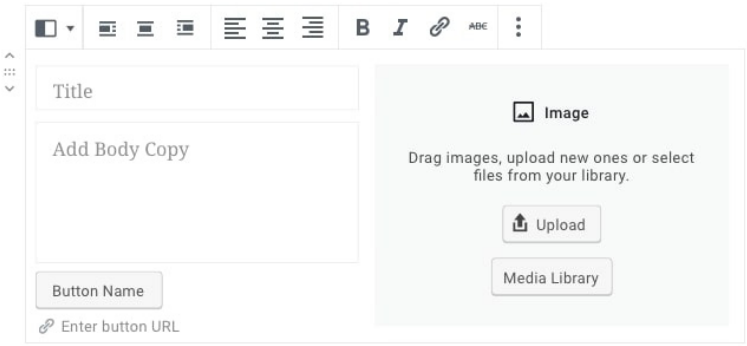
Now we will add some helper text for the URL. Insert text in your preferred way (by entering "T" on your keyboard, **Insert > Text** or the insert icon on the top left of the page). The last SketchPress feature to review is text styles. The WP5.0_Library is equipped with a variety of common text styles in the WordPress backend. To change the text to one of these styles, click on the text you wish to change. In the inspector panel, under Appearance, select the dropdown box for text styles. There are multiple options for typical WordPress text patterns. Select the one you see fit for this design.



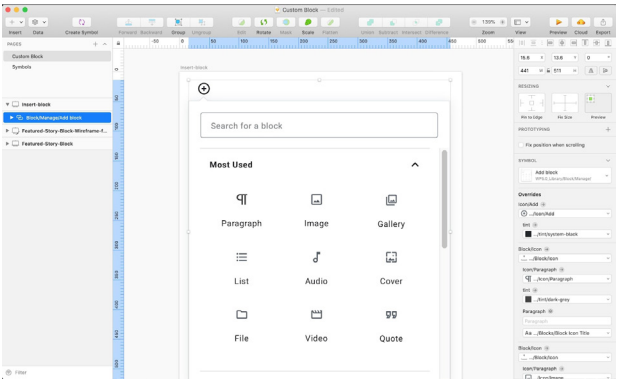

Your basic layout is now complete! This would certainly be enough to show any team member, so they could get a basic understanding of the design.



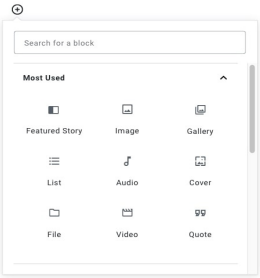
If you want to get more detailed, you can take it one step further by refining the block toolbar and including a new icon for the block itself. In this version, we added a new block tool bar **(Insert > WP5.0_Library > Component > Block–toolbar > Text)** and adjusted it to give the text formatting abilities.




SketchPress also has templates for inserting a block **(Insert > WP5.0_Library > Manage > Add Block)** which can be customised.



When you select the "add block" symbol, the inspector panel provides several options for changing the text and the icons for each of the block labels. You can create a custom symbol by adding your own icon or using one that is already in the SketchPress library. WordPress 5.0 uses Material Design Icons **[8]** for the backend, so there are plenty of resources.




A short video of this tutorial in action can be found on YouTube **[9].**

## Community support

The continued success of SketchPress is due in part to the contributors who want to make it the best possible resource for designers, engineers, and the WordPress community who need a quick way to define admin interfaces. Everyone is encouraged to contribute updates, fixes, and any general feedback in GitHub **[2].** There is also a Figma version **[10]** of SketchPress available, although it isn't maintained as regularly as the GitHub repo.

## Resources

[1] Focus tech and design leads
https://make.wordpress.org/core/2017/01/04/focus-tech-and-design-leads/

[2] 10up SketchPress Github
https://github.com/10up/SketchPress

[3] Design meeting notes 6.6.18
https://make.wordpress.org/design/2018/06/07/design-meeting-notes-for-june-6-2018/

[4] WordPress style guide colours
https://make.wordpress.org/design/handbook/design-guide/foundations/colors/

[5] WP5.0_Gutenberg.sketch
https://github.com/10up/SketchPress/blob/master/WP5.0_Gutenberg.sketch

[6] Get started with Sketch
https://www.sketchapp.com/docs/getting-started/

[7] Adding libraries to Sketch
https://sketchapp.com/docs/libraries/adding-libraries

[8] Material Design Icons
https://material.io/tools/icons/?style=baseline

[9] SketchPress custom block video tutorial
https://www.youtube.com/watch?v=V84a4qdtLQg&feature=youtu.be

[10] Figma version of SketchPress
https://www.figma.com/file/ZtN5xslEVYgzU7Dd5CxgGZwq/WP5.0_Library

The author

**Luc Princen**

Luc Princen is an independent developer and designer from the Netherlands specialising in WordPress and front-end development. He loves open source, elegant code, fast load times, Dungeons & Dragons and playing the occasional (grand) strategy game. You can find him on Twitter with the handle @LucP.

# Mastering Modularity: Building Gutenberg Blocks with JavaScript

by Luc Princen

With Gutenberg WordPress has made its first steps towards a modular approach to development. The idea is to reduce the amount of times you rewrite code across different projects. This is a completely different way of looking at development for WordPress, but it isn't a new idea. Plenty of frameworks and systems are working with a component-based approach.

WordPress's modular approach is written in JavaScript — another big shift from the original WordPress, which has traditionally been a PHP project.

This all adds up to a steep learning curve for WordPress developers, and it's understandable when people say they aren't going to invest time and resources in this as long as the classic version of WordPress is still being supported.

But I'm here to tell you that it will be worth it! A lot of the principles of both JavaScript and modular development aren't as hard or daunting as they might seem at first. The main reason to switch to something modular and modern is to eventually decrease the amount of rewriting you do on the tools you use in your projects. The entire goal of this switch is to save you time in the long run.

In this article I'll show you how to get started with building

Gutenberg blocks. By the time we're done, you will have created your first Gutenberg block, and you will understand the underlying logic of the Gutenberg ecosystem. So strap in, Dorothy, we're not in Kansas anymore!

## JavaScript and ES6

Let's start with the elephant in the room. If you've been writing PHP your entire professional life, switching to JavaScript might seem like the biggest chore. While I tend to agree with you, I find it hard to believe that you haven't worked on *anything* involving JavaScript. Ever since jQuery burst on the scene in the early noughties, JavaScript has been the tool to add subtle UI improvements for your visitors.

This is, however, very different from using JavaScript in a project-context. JavaScript has grown a lot the last couple of years. It isn't the spaghetti-code mess it was a decade ago. So in this chapter we'll be looking into every aspect of JavaScript that is relevant to our current goal: building a Gutenberg block. These aspects are:

- Getting to know modern Javascript (ES6)
- Setting up your project using Webpack and Babel

### Modern JavaScript, a crash-course

One of the best ways to get to know Gutenberg is to browse the uncompiled code, which is available on GitHub **[1],** but in order to do that we first need to take a small crash-course in modern

JavaScript or, as we will refer to it from now on, ES6.

> A note about versions:
> At the moment, new versions of JavaScript get released annually. While you might think that we'll be focussing on the latest-and-greatest in this article, most of Gutenberg was built using the 2015 (or ES6) standard, so we'll be using that as well. Apart from being the main version used in Gutenberg, it's also the term that gets the best results in Google for modern JavaScript.

There are many features and innovations we could list here that would improve your understanding of ES6, but there are three code examples that are important to get to the level of reading Gutenberg code yourself:

**Functions**

JavaScript is full of Closures — anonymous functions that get created to repeat a bit of logic. Therefore ES6 received an update in how we can create these anonymous functions a lot easier and quicker. Let's look at an example:

```
// An old anonymous function:
function( name ){
        return "Hello "+name;
}
```

```
// A new anonymous function, without a "return":
( name ) => { "Hello "+name }
```

Both of these functions will work in modern JavaScript. The second one is just a lot shorter. But let's look at the differences:

- There's no need to declare `Function()` anymore. Simply calling two round brackets followed by an arrow is enough.
- There's no `return`! ES6 automatically assumes that functions return something, so it will return the outcome of the last line of code by default.

You will see these functions a lot when browsing through Gutenberg-code. Be aware that this isn't some weird JavaScript voodoo magic, but a simple anonymous function.

**Variable types**

JavaScript has always had an issue with variables. Especially the variable **this,** which would shift and change meaning based on how deep your function was nested.

ES6 is trying to change that by introducing two new types of variables:
**let** and **const.**The big difference between these two is that **let** can be updated, while **const** can't. I'll give you a quick example:

```
var drink = 'Tea';
let food = 'Biscuit';
const place = 'Restaurant';
```

```
//First, let's look at var:
var drink = 'Coffee'; console.log( drink ); // -> 'Coffee'

//Let is a bit different:
let food = 'Cookie'; // -> Error: 'food' has already been
declared
food = 'Cookie'; console.log( food ); // -> 'Cookie'

//Const is even more rigid:
const place = 'Home'; // -> Error: 'place' has already been
declared
place = 'Home'; // -> Error: Assignment to constant variable
console.log( place ); // -> 'Restaurant'
```

In the example above we declare three types of variables: a **var,** a **let** and a **const.** The **var** can be redeclared and changed without any problems. The **let** returns an error when you try to fully redeclare it, but it still allows you to update it. And the **const** will remain the same value even if it's redeclared or updated.

Now you might say that the **var** offers the best flexibility and is therefore the easiest to use. In small titbits of code I would tend to agree with you, but if you're building most of your project in JavaScript, you'll need some constraints in order to not accidentally redeclare variables. If you browse the Gutenberg core code, you'll see plenty of **const** and **let** declarations and basically no **vars** because the project would simply become unwieldy.

### Destructuring

ES6 offers super-easy access to items within an object or array by adding destructuring declarations. The best way to show you how it works is by just diving into an example right away:

```
const person = {
        firstName: 'Johnny',
        lastName: 'Appleseed',
        age: 33,
        occupation: 'mechanic'
}

const { name: firstName } = person;
console.log( name ); // -> 'Johnny'
```

In this example we have an object called `person`. It contains a few key-value pairs. We'd like to take the `firstName` attribute out of `person`, so we can assign it to a different variable. Without having to loop through each key in `person` we can just say that the **const** name should be the value of `firstName` in `person`.

We can even use this without assigning a variable name right away. Consider this example:

```
const { firstName } = person;
console.log( firstName ); // -> 'Johnny'
```

If we don't mind changing the variable name, we can 'pluck' the first name out of the `person` object.

You will see this a lot in the Gutenberg project. WordPress has bundled specific components and functions in objects that you can reference in your code and add to your own blocks.

Like in this sample from an actual Gutenberg block:

```
const { MediaUpload, RichText } = wp.editor;
<RichText
tagName="p"
        value={value}
/>
<MediaUpload
        type="image"
        value={image}
/>
```

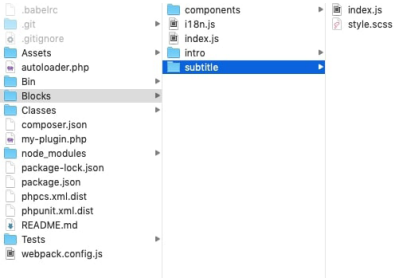
This example takes the `RichText` and `MediaUpload` components out of the globally available `wp.editor` object and allows you to use those components right away. I'll go deeper into using `RichText` later in this article, but for now be aware that those components are stored in `wp.editor` and that you can access them by using destructuring.

## Webpack, Babel and setting up a project

Working in a modular way means using many different files in your project. We will not be declaring multiple blocks in the same JavaScript-file because that would defy the reusability of your code, and we're here to save you time in the long run.

I tend to set up my Gutenberg projects using a dedicated /blocks folder in which I give all my blocks their own subfolder.

In that subfolder I can keep the main file that registers the block, but also the CSS styles, the SVG icon or any custom components this block might need. That way I have all the logic a specific block needs in its own folder, and I can just copy that block to a new project and get going.




Each block is in its own subfolder, each with its own stylesheet and other files it might need.

Unfortunately, though, JavaScript still runs on the client-side. This means all these files need to be downloaded and parsed correctly using the browser. This adds two major problems to our way working:

- More files means more latency, which means more load time. Ideally, we want one file for all of our blocks to reach the client.
- Supporting older browser versions is hard if we want to use modern JavaScript.

Making sure your code gets compacted together and is able to run in almost any environment and browser is, fortunately, something that can be largely automated.

For this we need Webpack (https://webpack.js.org) and Babel (https:/babeljs.io) .Other packages and frameworks are available, but Gutenberg-core uses these libraries, so let's use them as well.

While both packages are able to do a lot of things, their use in Gutenberg boils down to the following:

- **Webpack:** adds all components together in one file
- **Babel:** makes that one file understandable for every browser

There's plenty of information on both of these systems if you'd like to delve deeper into their inner-workings, but for now be aware that you'll need npm (https://www.npmjs.com/) and these three files to get it to work:

- **package.json [2]:** tells npm which tools to download, including Webpack and Babel
- **webpack.config.js:** tells Webpack how to deal with your files and where they are located
- **.babelrc:** tells Babel which settings you want to use

I've created a simple Gutenberg boilerplate project that has all of these things set up for you to get started. All you'll need to do after install is make sure you have npm installed and run two commands in your terminal program of choice:

`npm install` and `npm run dev`. This will allow you to start working on blocks. Get started with Gutenberg development by downloading this Gutenberg boilerplate plugin **[3].**

## Writing your block

Okay, so you've added our boilerplate plugin to the WordPress installation. You've also run `npm install` and **`npm run dev`** in the terminal and are now ready to create some blocks! In this chapter we'll focus on creating a simple intro block. This block will only do one thing: enable a user to create a special paragraph of text that's slightly larger than the regular paragraphs on the page. It might not sound very exciting, but by starting with the basics we'll quickly see how modular development works and how we can expand on this basic setup and create much more complex blocks.

### registerBlockType

Let's start at the beginning. Every block in Gutenberg gets registered using the `registerBlockType` function. This function is part of a global WordPress component called **`wp.blocks`** so, using destructuring in ES6 we can get access to this function like this:

```
//file: Blocks/intro/index.js
const { registerBlockType } = wp.blocks;
const name = 'myblock/intro';
const properties = {}

registerBlockType( name, properties );
```

This example registers a block to WordPress. I'll explain how we can customise our block a bit later (spoiler: the `properties` constant is about to get much more complex) but this is the basic setup.

## Enqueueing the script

The example above will try to add a block to the Gutenberg editor. But the script also needs to be loaded into WordPress. The boilerplate we're working with already handles this logic for you, but let's dive into it a bit, since WordPress added new hooks to accomplish this. Take a look at this example:

```
add_action( 'enqueue_block_editor_assets', function(){
        wp_enqueue_script(
                'myplugin-blocks',
                $url,
                [
        'wp-blocks',
        'wp-element',
        'wp-editor',
        'wp-i18n',
        'wp-components'
                ]
                );
});
```

This line tells the WordPress admin to load the JavaScript file located at `$url`, but more importantly it tells WordPress to load it after `wp-blocks, wp-element, wp-editor, wp-i18n` and `wp-components`. The reason for this is simple: We want to be able to use all of the Gutenberg goodies WordPress already has to offer by using destructuring, so we need to make sure they are loaded first.

## The attributes object

In the first example of this chapter we registered a block with the name `myblock/intro`. We've added a namespace to make sure it doesn't clash with blocks added by other plugins or themes. You might have tried this code yourself and found that it produces JavaScript errors all over the place. This is because of the mysterious `properties` variable, which is empty in our example.

The `properties` variable contains most of the logic in a Gutenberg block and really can't be empty. Let's look at a basic example of a `properties` object, piece-by-piece:

```
const properties = {
        title: "Intro",
        description: "An awesome intro block",
        category: "common",
        icon: 'megaphone',
        attributes: {
                intro_text: {
                        type: 'array',
                        selector: '.intro',
                        source: 'children',
                        default: 'This is an intro
        text'}},
        edit( props ){
                return null;},
        save( props ){
                return null;
}}
```

This example still won't work, because we've left the `edit` and `save` attributes empty for now. We'll focus on the other stuff first.

In this example we tell Gutenberg that we want to add a block called "Intro". We've already passed Gutenberg a name for this block with a namespace, but this name is what will show up in Gutenberg's UI. It's the user-friendly version of the name of our block.

We also pass a simple description and a category. The category-attribute has some options which I encourage you to google, but 90 percent of the time you'll be using `common`. The `icon` attribute tells Gutenberg which icon to display in the block-selector. This can be the name of a Dashicon or the raw SVG markup of another icon — it's your choice.

Then we get to the most interesting part of our `properties` object: `attributes`. This tells Gutenberg what kind of fields it can expect in this block and how to distil the value of this field. Attributes come in many different shapes and sizes, but overall the types you'll most likely use are the `array`, `string` and `number` types. Let's break this example down into code comments:

```
attributes: {

        //this block has a variable named intro_text
        intro_text: {
        //intro_text can contain text, html and
        links. We need to//save this as an array:
        type: 'array',
```

```
            //you can find the contents of this block in the
            .intro
            //html-tag:
            selector: '.intro',

            //grab all the children in .intro
            source: 'children',

            //the default value of this variable is this
            text:default: 'This is an intro text'
            }
    }
```

So we create a variable called intro_text that contains HTML. We'll save it with the classname of `.intro`, and its value will be all the children of `.intro`. Except when the value is empty, then we display a default text.

By assigning a block's attributes in advance, Gutenberg can check if nothing funky is going on with the data it's trying to save. This prevents possible security problems but also forces you to think ahead of what type of block you're trying to build.

## The edit() function

In the `properties` example we added an `edit()` and a `save()` parameter, which are both functions. We made them return `null` for the time being, but now it's time to put both of these to work. Simply put:

> edit() is what people get to see in the WordPress admin.
> save() is what gets put out to the front-end of your site.

So we're creating the `edit()` function for our intro block. We said that this block should be able to create a paragraph of slightly larger text. So we need a way to input text first. Once we've added a way to input our text, we'll figure out a way to add a custom class in the `save()` function.

Let's look at the following example:

```
//start of the document:
const { RichText } = wp.editor

//in properties:
edit( props ){

        //get intro_text and setAttributes out of props:
        const { attributes: { intro_text },
        setAttributes } = props;
```

```
            //allow intro_text to be saved:
            const onChangeIntro = ( value ) => {
            settAttributes( { intro_text: value } )
            }

            //return the actual HTML:
            return (
            <RichText
                    tagName="p"
                    className="intro"
                    onChange={onChangeIntro}
                    value={intro_text}
                    />
            );
    }
```

There's a lot happening in this example! We fetch the value of `intro_text` out of the `props` variable. We also fetch the `setAttributes` function out of the props variable. This `props` variable is provided by Gutenberg out-of-the-box and basically contains every bit of information WordPress has on your block at the moment.

Both `edit()` and `save()` are functions that get updated automatically once something in `props` changes. So if we change the intro in `<RichText>`, onChange triggers the `onChangeIntro` function, which uses `setAttributes` to set the `intro_text` to the new value entered. Once `setAttributes` gets called, Gutenberg saves the data, re-renders the block on the backend (using `edit()`) and on the frontend (using `save()`) with the updated values. To put it simply: each keystroke, each tiny change

to the contents of a block gets saved automatically and triggers a re-render on both the frontend and the backend.

This is important to keep in mind, because your `edit()` and `save()` functions will almost always be working with live-changing data.

## The RichText Component

In our example we're using one of the many components Gutenberg already ships with: the `RichText` component. The idea behind this component is to remove as much UI as possible when entering text. Traditionally, when you're entering text into a web-based application, you'd do so in `<input>` fields. Even the original WordPress WYSIWYG editor looked like a regular form field. The `RichText` component changes that by basically allowing you to update the HTML tags themselves.

```
<RichText
        tagName="p"
        className="intro"
        onChange={onChangeIntro}
        value={intro_text}/>
```

Notice the first attribute of this `RichText` component: we tell it to represent an ordinary `<p>` tag. As you might have guessed; this `tagName` attribute can be pretty much anything you want. It makes editing in Gutenberg a very seamless experience. It's called the RichText component because it allows you to add bold or italic text and even links inside it.

If you want fine-grained control over the types of formatting this component offers, you can use `formattingControls`:

```
<RichText
        tagName="p"
        formattingControls=['bold', 'italic',
'strikethrough', 'link']
/>
```

The `RichText` component does more than make sure you can input your data in a clean way. It also helps you to render that content properly, which we'll see in the final piece of this block-building tutorial.

## Saving your data

Much like the `edit()` function, the `save()` function dictates the markup of the block. Whereas `edit()` is meant for the WordPress back-end, `save()` is meant for the website's front-end. Continuing with our intro block example, the output of an intro block may look something like this:

```
save( props ){
        //fetch the intro_text:
        const { attributes: { intro_text } } = props;

        return (
                <div className="intro">{intro_text}<div>
        );
}
```

This example will output a div with the class of "intro" and the `intro_text` inside that div. The reason we're using `className` as attribute instead of `class`, by the way, is because the word `class` is part of the JavaScript language. Using it would probably upset a tool like Babel, as it tries to make sense of your code.

We need the class .intro, because when we declared our attributes, we told the attributes object to look for the contents of `intro_text` in the selector `.intro`:

```
attributes: {
        intro_text: {
                type: 'array',
                selector: '.intro',
                source: 'children',
                default: 'This is an intro text'
        }
},
```

Without that class our data would fail to save, and Gutenberg wouldn't be able to find the contents typed in `RichText`.
A better way to handle this output is by using the `RichText`'s "Content"-feature. We can use `RichText` again to render the content as-is, without needing a wrapping div but with the possibility of telling our `save()` function that it should be save with a specific `className`. So, expanding on our precious save-function, this would be the final result:

```
        save( props ){
                const { attributes: { intro_text } } =
props;
                return (
                        <RichText.Content
                                className="intro"
                                tagName="p"
                                value={ intro_text }
                        />
                );
        }
```

Which would output the following HTML:

```
<p class="intro">
        This is text I filled into our <b>RichText</b>
component.
        <a href="https://github.com/WordPress/gutenberg/tree/
master/packages/editor/src/components/rich-text">Read more
about it here</a>
</p>
```

The entire JavaScript file of this simple intro block can be downloaded from GitHub **[4].** Feel free to experiment with this example.

## Conclusion

So, with a little bit of work we've created a simple block. We've seen the power of the `RichText` component and we can begin to gradually increase the complexity in our blocks, for instance by adding a `MediaUpload` component, to add a simple image to our intro.

One of the greatest resources on Gutenberg is the Gutenberg project on GitHub **[1].** You'll find plenty of documentation on how to create your own blocks and use WordPress' native components, but it will also show you proper ways of working with the Gutenberg ecosystem. Another great source of Gutenberg information is Zac Gordon's Gutenberg course **[5].** While this is a resource you'd have to pay for, I definitely recommend you taking on that investment.

If you have any questions regarding this article or Gutenberg in general, don't hesitate to look me up on Twitter @LucP.

## Resources

[1] Gutenberg uncomplied code
https://github.com/wordpress/gutenberg

[2] package.json
https://docs.npmjs.com/files/package.json

[3] Gutenberg boilerplate plugin
https://github.com/lucprincen/my-plugin

[4] Gutenberg intro block
https://github.com/lucprincen/gutenberg-intro-block

[5] Zac Gordon's Gutenberg course
https://javascriptforwp.com/product/gutenberg-block-development-course/

## The author
## **Daniel Schutzsmith**

Daniel Schutzsmith (daniel.schutzsmith.com) is a rare breed – a hybrid of equal parts design, code, and strategy. He's devoted his career to making positive change to protect our world for generations to come.

Currently, Daniel is the digital technology director of Natural Resources Defense Council (NRDC), leading an amazing team to create websites, apps and interactive experiences to help safeguard the earth – its people, plants and animals, and the natural systems on which all life depends. Previously, Daniel was the digital technology manager / senior web developer for Amnesty International USA.

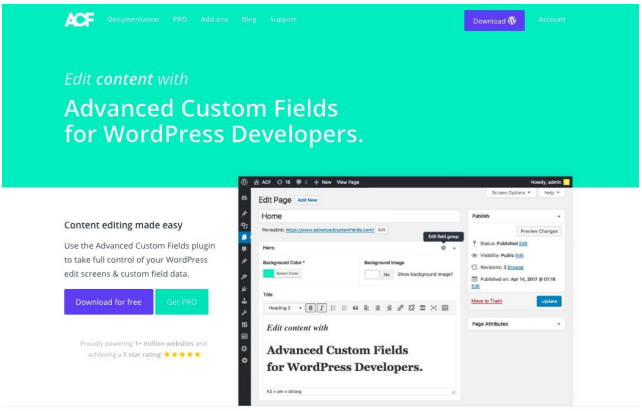
# Building Gutenberg Blocks with Advanced Custom Fields

by Daniel Schutzsmith

In Heart Internet's Design Systems **[1]** book I shared a case study of how Amnesty International USA used Advanced Custom Fields (ACF) (advancedcustomfields.com) with "Flexible Fields" and "Repeater Fields" to achieve dynamic page layouts without the use of any page builders. The simplicity and ease of use of Advanced Custom Fields is wonderful for WordPress administrators and authors alike.

Since then, WordPress has debuted the new Gutenberg editor, integrated into all builds beyond WordPress 5.0.

Gutenberg is a fantastic tool that gives users the capability to lay out their webpages without needing to know code and with the ease of drag and drop. Given its young age, there is still much room for it to evolve and mature throughout the industry.




The Advanced Custom Fields website

One of the challenges developers have been faced with is the process of creating new blocks, which so far has been mostly achieved through JavaScript-based solutions. These solutions are often extremely complicated and present a much higher learning curve to WordPress developers who tend to use JavaScript mostly for front-end, not back-end, needs.

You can create your own blocks for Gutenberg using new modern JavaScript methods (see the previous chapter) but frankly, most WordPress developers have been building with PHP from the backend. This shift in coding methodologies has frustrated much of the industry, which is searching for solutions that are easy to understand and implement.

Thankfully the folks behind Advanced Custom Fields have come up with a solution!

## ACF to the rescue

In October 2018, Advanced Custom Fields announced a new capability to create Gutenberg blocks with the release of ACF 5.8 **[2].**

ACF's implementation attempts to make the creation of new Gutenberg blocks a simple and familiar process that any WordPress developer will feel comfortable with, as it uses primarily HTML and PHP.

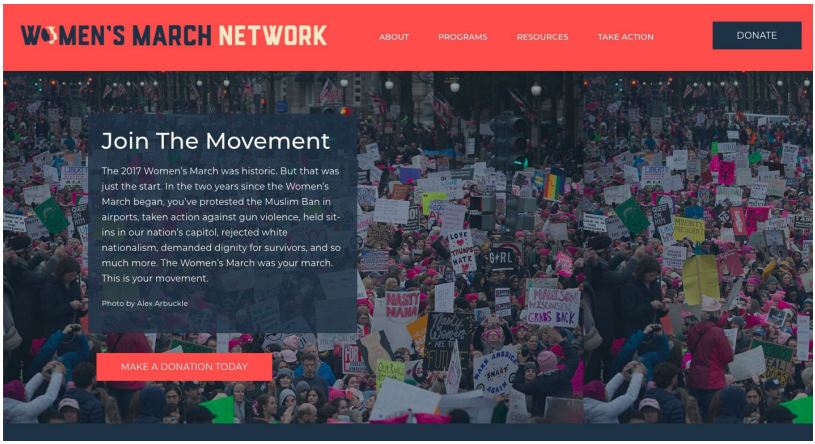
Before we dive in, I will acknowledge that this solution is not for everyone. If you enjoy writing JavaScript, or are looking to create a plugin to publish, then you'll want to use a solution closer to something like Ahmad Awais' Guten Block Template **[3].**

With that disclaimer out of the way, let's get to it.

## Hero block example

In this example, I'll show you how I created a new Hero block for the Women's March Network website (womensmarch.org) using PHP, HTML, and CSS.

There are only three steps to follow, and each step outlines a unique piece in the process. First, we'll register the block. Second, we'll create the group and fields in ACF. Finally, we'll place the code we want to show up in the preview and on the front-end into a template.




The homepage hero block used on the Women's March Network website.

## Step 1: Register the block

Before we create any custom fields, we first need to register the block in our theme's **functions.php** file just like we would for a widget or custom post type. We'll use the new `acf_register_block()` to do this.

I know it feels weird to do this first instead of creating the field group and custom fields in ACF, but if we didn't register the block first, then we'd have no way for ACF to associate it to a field group.

```
/**
 * Custom Gutenberg Blocks using Advanced Custom Fields
 */
add_action('acf/init', 'wm_acf_init');

function wm_acf_init() {

        // check function exists
        if( function_exists('acf_register_block') ) {

                // register a block
                acf_register_block(array(
                'name'          => 'wm_hero',
                'title'         => __('Hero', 'wm'),
                'description'   => __('A custom hero block.',
'wm'),
                'render_template'=> 'gutenberg-templates/wm_
hero.php',
                'category'      => 'layout',
                'icon'          => 'slides',
                'keywords'      => array( 'hero', 'header',
'background' )
                ));
        }
}
```

Code to register the homepage hero block for the Women's March Network.

As ACF points out in their blog post **[2],** these settings are closely related to the WordPress JavaScript `registerBlockType()` function **[4].**

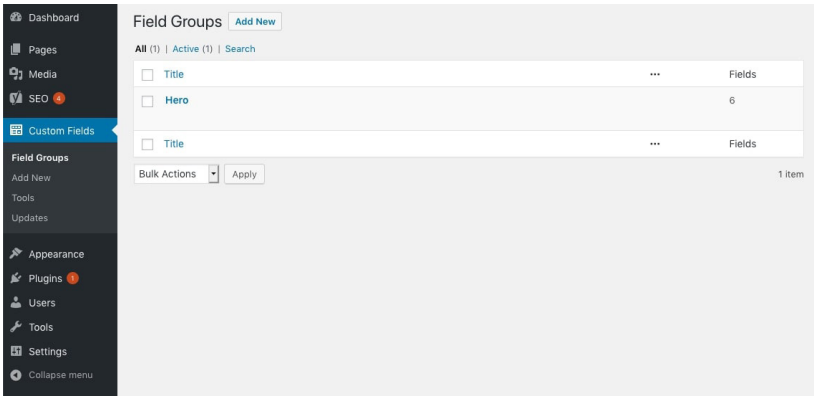
The settings array provides you with similar elements to other common WordPress functions like widgets and menus. Here is the breakdown of each setting:

- **Name:** The unique name for the block to be used in code only. A block name can only contain lowercase alphanumeric characters and dashes, and must begin with a letter. Also, please note that this name will be used in the comments that Gutenberg writes to define where it should appear.
- **Title:** Shown as the display title for the block used in the editor. The block inserter will show this name.
- **Description:** Shown as the brief description used in the editor. This will be displayed in the block inspector.
- **Render_Template:** The path to a template file used to render the block HTML. This can either be a relative path to a file within the active theme or a full path to any file.
- **Category:** Blocks are grouped into categories to help users browse and discover them. The core provided categories are: common, formatting, layout, widgets, and embed. Custom block categories can also be created following this method **[5].**
- **Icon:** The icon will show up in the block inserter to make it easier to identify a block. These can be any of WordPress' Dashicons **[6]**, or a custom SVG element.
- **Keywords:** A block can have aliases that help users discover it while searching. One thing to note is that more than three aliases keeps the block preview from showing correctly in the editor.

A comprehensive list of all possible block settings can be found in the Gutenberg Handbook **[7].**

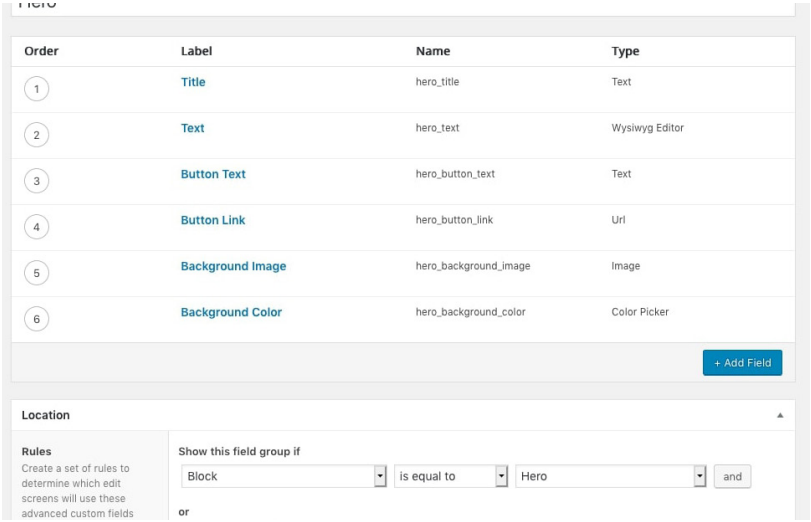
## Step 2: Create the Field Group and Fields

Now that we have the block registered in our functions.php, we'll create a new Field Group in the Advanced Custom Fields admin panel. In our case, I've created a Hero Field Group as seen below.




Field Group created in Advanced Custom Fields

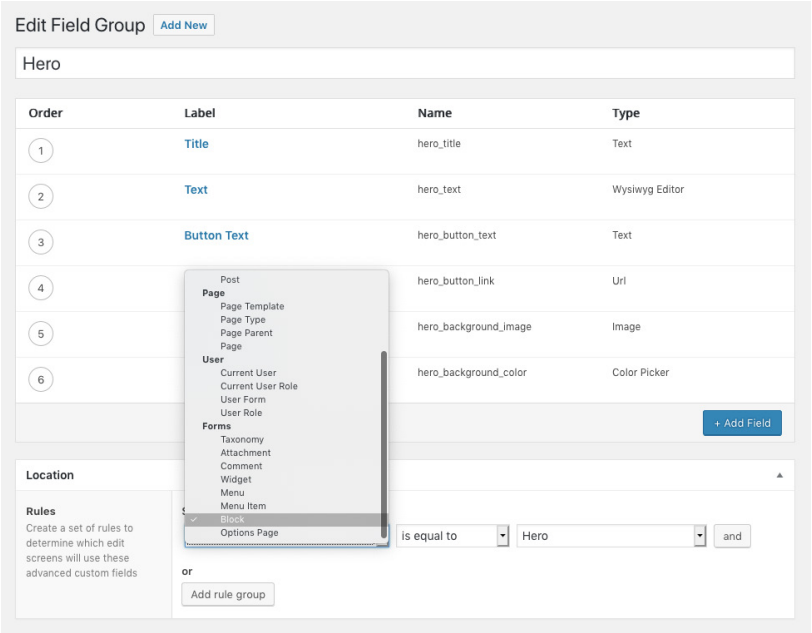
Inside of the Field Group, we'll then add all of the fields that you want the user to be able to provide values for. In our Hero example, we're allowing them to edit the text, button, and background that we'll use in the block.




Fields for our Gutenberg Block in Advanced Custom Fields.

Finally, to associate the Field Group with the block, we need to update the rules setting. In the drop-down you should see the title of the block you registered in step 1.

If your block doesn't show up, then make sure you didn't miss




something in step 1.

Choose ‘Block’ as the type of rule and then select your block that you registered in Step 1

## Step 3: Create the block template

Our last step is to create the template that will hold our PHP, HTML, and CSS.

The location of this template should be whatever you specified in the Render_Template value in Step 1. If you made it a relative path, as in our example, then you'll want to place this file in the proper location. In our case it was inside a folder I created called 'gutenberg-templates' and the file is named 'wm-hero.php'. You can name these whatever you want, there are no hard rules. I suggest following whatever naming convention makes sense based on the theme type you are using. For the Women's March Network website the theme was Understrap (understrap.com).

As you'll see in the example below, this template looks very similar to a content or page template that you'd have in your theme. Basically, we're loading in the fields that the user has specified, hiding ones that were not, and also adding some CSS that is specific to this block and this block only.

```
<?php
/**
 * Hero template for Gutenberg
 *
 * Built on top of Bootstrap and Advanced Custom Fields
 *
 * @package understrap
```

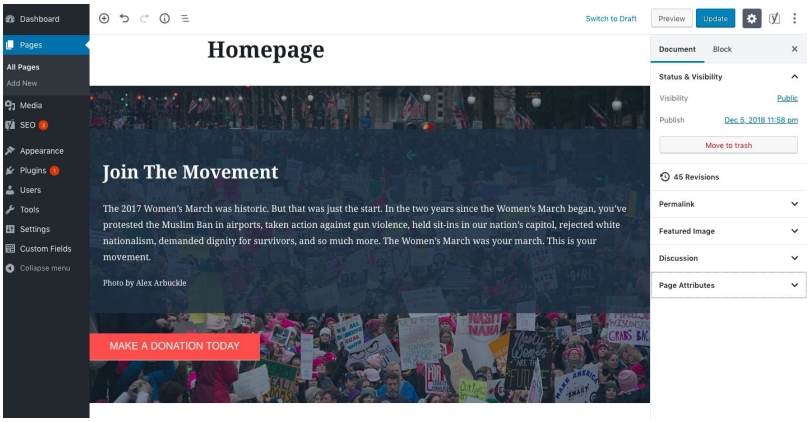
```
 */
// Load fields.
$title            = get_field( 'hero_title' );
$text             = get_field( 'hero_text' );
$button_text      = get_field( 'hero_button_text' );
$button_link      = get_field( 'hero_button_link' );
$background_image = get_field( 'hero_background_image' );

// create id attribute for specific styling
$id = 'hero-' . $block['id'];

?>
<section class="hero-block" id="<?php echo $id; ?>">
        <div class="container">
                <div class="row">
                        <div class="hero-content col-md-5 col-
sm-12">
                                <?php if ( $title ) : ?>
                                        <h1 class="hero-
title"><?php echo esc_html( $title ); ?>
</h1>
                                <?php endif; ?>
                                <?php if ( $text ) : ?>
                                        <p class="hero-
description"><?php echo $text; ?></p>
                                <?php endif; ?>
                        </div><!-- .hero-content-->
                </div>
                <div class="row">
```

```
        <div class="hero_button col-md-5 col-sm-12">
                                <?php if ( $button_text &&
$button_link ) : ?>
                                        <a class="button button-
hero btn btn-primary" href="<?php echo esc_url( $button_link );
?>"><?php echo esc_html( $button_text ); ?></a>
                                <?php endif; ?>
                        </div>
                </div>
        </div>
</section><!-- .hero -->
<style type="text/css">
        #<?php echo $id; ?> {
                background: url('<?php if ( $background_image )
: ?><?php echo
                 $background_image; ?><?php endif ?>');
                padding: 75px 0;
        }
        #<?php echo $id; ?> .hero-content {
                background: rgba(33,51,68,.8);
                padding: 24px;
                margin-bottom: 35px;
                color: #fff;
        }
        #<?php echo $id; ?> .hero-content h1 {
        }
</style>
```

Code for the Hero block template for Women's March Network.




Now that the template is complete, we will see it working on the front-end and also be able to preview it in the Gutenberg editor.

## Taking it further

That's it! In three simple steps we've made a new Gutenberg block. Now you can use this process to create your own blocks for your websites with Advanced Custom Fields.

For further customisation I'd suggest checking out the Designer & Developer Gutenberg Handbook.

Happy building!

## Resources

[1] Heart Internet's Design Systems Ebook
https://www.heartinternet.uk/blog/design-systems-free-ebook-download/

[2] ACF blocks for Gutenberg
https://www.advancedcustomfields.com/blog/acf-5-8-introducing-acf-blocks-for-gutenberg/

[3] Guten block template
https://github.com/ahmadawais/create-guten-block

[4] Writing your first block type
https://wordpress.org/gutenberg/handbook/blocks/writing-your-first-block-type/

[5] Managing block categories
https://wordpress.org/gutenberg/handbook/designers-developers/developers/filters/block-filters/#managing-block-categories

[6] WordPress Dashicons
https://developer.wordpress.org/resource/dashicons/

[7] List of all possible block settings
https://wordpress.org/gutenberg/handbook/designers-developers/developers/block-api/block-registration

# Your business deserves great WordPress hosting

## INSTALL WORDPRESS IN SECONDS

Get the latest version installed with just one click – no need to download files, set up databases, or change permissions

## HOSTING WITHOUT LIMITS

Our hosting packages have unlimited storage, bandwidth, and databases as standard.

## UNRIVALLED PERFORMANCE AND SPEED

We only use the latest hardware and software on our platform to give you a fast and responsive website.

## ALWAYS HERE TO HELP

In-house Customer Services team is here to help you with any questions - 24/7/365.

Find your perfect hosting package at www.heartinternet.uk or call us on 0330 660 0255