The R Markdown team is happy to share that **blogdown** version 1.0 is now available on CRAN. **blogdown** was originally released in the fall of 2017. The latest version of the package includes some significant improvements to the user experience, and some under-the-hood improvements that you'll benefit from without even knowing!

Latest release

Last bookdown release 1.0 cran badge

You can install the latest version from CRAN:

```
install.packages("blogdown")
```

In this post, we'll share some highlights from the latest release, but you might want to look at the release notes for the full details.

Workflows

If you are already a **blogdown** user, you will notice some important changes in how the package works. Previously, **blogdown** did two things automatically for you if you had started serving your site as you worked:

1. Knitting R Markdown files upon save.
2. Re-knitting already knitted R Markdown files, based on a timestamp filter.

While both of these behaviors were sometimes helpful, we found that more often than not they were problematic for users. Based on user feedback, we have provided options to disable them. The second behavior is disabled by default now (more on this later). For the first behavior, you can set the global option `options(blogdown.knit.on_save = FALSE)` to disable it. After that, you must knit an R Markdown post with intent to see your edits take effect (note that plain Markdown files with the extension `.md` do not need to be knitted).

The really great news is that the “Knit” button now *works* for **blogdown** content in the RStudio IDE! Please feel free to retrain your fingers to knit.

If you have not yet served your site (after using the “Serve Site” addin or `blogdown::serve_site()`), clicking on the “Knit” button will start the server automatically and produce the site preview for you.

Also, the `public/` directory will no longer be generated when you serve the site. Where did it go? We are now using Hugo’s server, which means the website is not rendered to disk by default, but instead served directly from memory. The Hugo server is much faster, and also supports navigating to the output web page of which you are currently editing the source document.

If you miss the `public/` folder and want it back, you’ll need to build the site explicitly via `blogdown::build_site()`, or if you use RStudio, press `Ctrl + Shift + B` or `Cmd + Shift + B` to build the website. This function no longer recompiles R Markdown files by default, because it may be expensive and often undesirable to compile `.Rmd` files that have been compiled before.

If you do want to do that anyway, `build_site(build_rmd = TRUE)` will recompile *everything* (look out!). For easier control, this `build_rmd` argument can also take a function to filter the files to re-render when building the site. We have introduced 3 helpers functions (and you can use your own) with an alias each for ease of use:

- `build_site(build_rmd = 'timestamp')` uses `filter_timestamp()` and will compare the timestamp of input and output files to decide which to render.
- `build_site(build_rmd = 'newfile')` uses `filter_newfile()` and will render only files that have no output file yet.
- `build_site(build_rmd = 'md5sum')` uses `filter_md5sum()` and will compare MD5 checksums of files to decide to render.

See the help page `?blogdown::build_site` for more information.

Checking functions

blogdown 1.0 comes with new *check* functions to help you diagnose and prevent build issues with your site. Checks will help you identify known issues and provide opinionated recommendations to guide you into the pit of success.

There are 5 specific `check_*` functions:

- `check_config()` checks the configuration file (`config.yaml` or `config.toml`).
- `check_gitignore()` checks if necessary files are incorrectly ignored in GIT.
- `check_hugo()` checks possible problems with the Hugo installation and version.
- `check_netlify()` checks some important Netlify configuration `netlify.toml`.
- `check_content()` checks for possible problems in the content files, like the validity of YAML metadata, some posts with future dates and draft posts, or R Markdown posts that have not been rendered.

A final function, `check_site()`, will run all above `check_*()` functions at once. If you are a blogdown educator, you may go step-by-step with the checking functions to help students gain a mental model of all the moving pieces needed to build and deploy a site. For people familiar with GitHub, Netlify, and Hugo, you may want to just check everything with `blogdown::check_site()`.

These functions will show you what is checked, why, and will assign you some `[TODO]` items that need your action.

```
------------------------------------------------------------
○ A successful check looks like this.
● [TODO] A check that needs your attention looks like this.
| Let's check out your blogdown site!
------------------------------------------------------------
```

We hope you'll find the checks as helpful as several other users have. We'll continue to incorporate more checks into these functions in the future. When in doubt, try `remotes::install_github('rstudio/blogdown')` and see if `blogdown::check_site()` uncovers any new problems.

As an extra bonus, as we were working on better messaging for these `check_*()` functions, we also improved the new site experience when running `blogdown::new_site()`. **blogdown** will now output more user-friendly messages on what is going on during your new site setup. To follow a complete "code-through" of setting up a new site with the new **blogdown**, go look at Up & running with blogdown in 2021 written by Alison Hill.

Hugo versioning system

Although **blogdown** also supports Jekyll and Hexo, most users power their websites with the Hugo static site generator. Hugo has a lot of pluses, one of which is that it gives you fast site builds. However, one minus we've found as experienced users over the past 3 years is that Hugo also changes fast—new functions are added and deprecated, and it can be difficult to keep track of if you have more than one Hugo site. This can lead to frustration trying to debug why a site that you could build last month will not build now.

blogdown now gives you a way to pin your website project to a specific Hugo version. Both `blogdown::install_hugo()` and `blogdown::check_site()` will tell you how. You may also use the following to find all your locally installed Hugo versions:

```
blogdown::find_hugo('all')
```

You'll see the versions that you have available like this:

```
[1] "/Users/alison/Library/Application Support/Hugo/0.54.0/hugo"
[2] "/Users/alison/Library/Application Support/Hugo/0.71.1/hugo"
[3] "/Users/alison/Library/Application Support/Hugo/0.78.2/hugo"
[4] "/Users/alison/Library/Application Support/Hugo/0.79.0/hugo"
[5] "/usr/local/bin/hugo"
```

From these available Hugo versions, if you'd like to pin a specific one to a particular project, you'll use a project-level `.Rprofile` file. You may call this new helper function to create and fill the `.Rprofile` with recommended **blogdown** options:

```
blogdown::config_Rprofile()
```

Inside that file, to pin Hugo to the version, say, 0.79.0, you may set:

```
options(blogdown.hugo.version = "0.79.0")
```

Note that you must restart your R session for changes in your `.Rprofile` file to take effect. How could `check_site()` or `check_hugo()` help you do all this? Let's check it out:

```
blogdown::check_hugo()

― Checking Hugo
| Checking Hugo version...
○ Found 4 versions of Hugo. You are using Hugo 0.79.0.
| Checking .Rprofile for Hugo version used by blogdown...
| Hugo version not set in .Rprofile.
● [TODO] Set options(blogdown.hugo.version = "0.79.0") in .Rprofile.
● [TODO] Also run blogdown::check_netlify() to check for possible
problems with Hugo and Netlify.
― Check complete: Hugo
```

Now, as we hint above in a `[TODO]` item, after you've pinned a project-level Hugo version, you'll want to ensure that your Hugo version used by Netlify to build your site also matches your local version. Again, the checking functions `check_netlify()` can help you here, but you may also use:

```
blogdown::config_netlify()
```

To open and edit that file with your updated Hugo version number. After doing that, if we checked this file, we'd see:

```
blogdown::check_netlify()

― Checking netlify.toml...
○ Found HUGO_VERSION = 0.79.0 in [build] context of netlify.toml.
| Checking that Netlify & local Hugo versions match...
○ It's a match! Blogdown and Netlify are using the same Hugo version
(0.79.0).
| Checking that Netlify & local Hugo publish directories match...
○ Good to go - blogdown and Netlify are using the same publish
directory: public
```

```
— Check complete: netlify.toml
```

You may also want to periodically clean up your older Hugo versions that are no longer in use. To do this, use:

```
blogdown::remove_hugo()
```

In your console, you'll see an interactive menu that allows you to choose which versions to remove like this:

```
------------------------------------------------------------
--------------------
5 Hugo versions found and listed below (#1 on the list is currently
used).
Which version(s) would you like to remove?
------------------------------------------------------------
--------------------

1:   /Users/alison/Library/Application Support/Hugo/0.54.0/hugo
2:   /Users/alison/Library/Application Support/Hugo/0.71.1/hugo
3:   /Users/alison/Library/Application Support/Hugo/0.78.2/hugo
4:   /Users/alison/Library/Application Support/Hugo/0.79.0/hugo
5:   /usr/local/bin/hugo

Enter one or more numbers separated by spaces, or an empty line to
cancel
```

If you want to update Hugo, you'll need to install a new version now using `install_hugo()` and a specific version. By default, it will install the latest available version. Consequently, the previous `update_hugo()` function has been deprecated.

Page bundles

Hugo version 0.32 introduced a new feature called "Page Bundles," as a more natural way to organize your content files. The main benefit of using page bundles instead of normal pages is that you can put resource files associated with the post (such as images and data files) inside the same directory as the post itself. This means you no longer have to put them under the `static/` directory, which has been quite confusing to Hugo users. Here is an example of two page bundles, both inside the `content/post` section:

```
.
└── content
    ├── post
    │   ├── raindrops-on-roses
    │   │   ├── index.md // That's your post content and Front Matter
    │   │   └── assets
    │   │       ├── rain.jpg
    │   │       ├── roses.jpg
    │   │       └── thorns.csv
    │   └── whiskers-on-kittens
    │       ├── index.md // That's your post content and Front Matter
    │       └── assets
    │           └── kittens.jpg
    └── songs
```

blogdown now works better with page bundles, like `raindrops-on-roses` and `whiskers-on-kittens` in the above example. The "Insert image" and "New post" add-ins work, and when you knit your posts or other R Markdown-based content, all figures and any other dependencies (like the `index_files/` and `index_cache/` folders) will be output to your page bundle instead of to a folder nested in the `static/` directory. Consequently, you should not ignore `"_files$"` in the `ignoreFiles` field in your site configuration file. This will also get flagged for you if you run `blogdown::check_site()`.

If you prefer no bundles, you may set `options(blogdown.new_bundle = FALSE)` in your `.Rprofile` to get the old behavior back.

Finally, if your pages have not been bundled up yet, we have provided a new helper function `bundle_site()` to help you convert normal pages to bundles. Unfortunately, **blogdown** v1.0 contains a small bug that we failed to discover (sorry about that), so if you want to try out the conversion, we recommend that you try the development version instead:

```
remotes::install_github("rstudio/blogdown")
# restart R, and make sure your project is either backed up or under
version control
blogdown::bundle_site(".", output = ".")
```

Better support for Markdown format

blogdown lets you work with three formats for your website content, each of which is processed and rendered slightly differently:

File format	Processed by	Output format	Additional processing by
`.Rmd`	→ Pandoc	→ `.html`	
`.Rmarkdown`	→ Pandoc	→ `.markdown`	→ Hugo / Goldmark
`.md`	→	→ `.md`	→ Hugo / Goldmark

To learn more about these formats, you may read this blogdown book chapter.

Of course, you may always use `.md` for content that does not include any R code—this content will only be processed by Hugo's Markdown renderer (the default now is Goldmark). Content written in `.Rmd` files will be rendered with Pandoc straight to `.html`, bypassing Hugo's markdown renderer completely. However, some Hugo themes depend on Markdown files as input (not `.html`)—that is why the `.Rmarkdown` file extension has existed. `.Rmarkdown` files are rendered to `.markdown` using **knitr** and Pandoc, which will then be processed by Hugo Markdown renderer. This special file extension (unique to **blogdown**) means that the file will be processed by R before Hugo, allowing you to use R code, which a plain `.md` will not allow.

With **blogdown** 1.0, the `.Rmarkdown` file format has become more fully featured, to help users take better advantage of some Hugo theme features and configuration options. `.Rmarkdown` files now support bibliographies and HTML widgets like a standard `.Rmd` document. This makes `.Rmarkdown` an interesting format to be used alone in **blogdown** projects. However, some users voiced a need to be able to simply keep the `.md` version of their `.Rmd` content, without needing to use the special `.Rmarkdown` file extension.

blogdown now offers a build method to render `.Rmd` files to `.md` instead of `.html`. This special *full markdown mode* can be activated by setting `options(blogdown.method = "markdown")` in your `.Rprofile`.

We recommend this `"markdown"` mode to advanced users who have a high comfort level with Hugo, and want to use the full power of Goldmark (and understand the trade-offs of not using Pandoc for rendering here, e.g., not all Pandoc's Markdown features are supported by Goldmark).

Final notes

This version is a big milestone for **blogdown**, with a lot of changes and improvements. Some improvements may not even be noticeable, yet they are important. For example, Hugo requires you to install GIT and Go if you use themes that contain "Hugo modules," but we don't wish to turn **blogdown** users into Go developers, so we tried hard to get rid of the dependency on GIT and Go in this case. Similarly, multilingual sites are better supported under the hood now.