# Contributing to Timber

Hey there! We’re really happy you’ve found your way here ?. Timber is a community project that is developed by people from all over the world. We appreciate any help.

Here are ways to get involved:

1. [Star](https://github.com/timber/timber/stargazers) the project!

2. Answer questions that come in through [GitHub issues](https://github.com/timber/timber/issues?state=open).

3. [Report a bug](https://github.com/timber/timber/issues/new) that you find.

4. Share a theme you’ve built with Timber. This helps transfer knowledge about best practices, etc. \_Add it to the [Showcase list](https://github.com/timber/timber/wiki/Showcase)\_.

5. Tweet and [blog](http://www.oomphinc.com/blog/2013-10/php-templating-wordpress/#post-content) about the advantages (and criticisms) of the project and Twig.

6. Browse ["help wanted"](https://github.com/timber/timber/labels/help%20wanted) and ["good first issue"](https://github.com/timber/timber/labels/good%20first%20issue) labels for areas of WordPress/PHP/code you know well to consider, build or document.

7. Answer questions on [Stack Overflow posted under the «Timber» tag](https://stackoverflow.com/questions/tagged/timber). You can also [subscribe to a tag](https://stackoverflow.blog/2010/12/20/subscribe-to-tags-via-emai/) via email to get notified when someone needs help.

8. Answer questions in the support channel on [Gitter](https://gitter.im/timber/timber).

## Table of Contents

<!-- TOC depthTo:3 -->

- [Pull Requests](#pull-requests)

- [Coding Standards](#coding-standards)

- [Use PHP\_CodeSniffer to detect coding standard violations](#use-php\_codesniffer-to-detect-coding-standard-violations)

- [Inline Documentation](#inline-documentation)

- [Differences to the official standards](#differences-to-the-official-standards)

- [Ignoring Structural Elements](#ignoring-structural-elements)

- [Referencing class names](#referencing-class-names)

- [Code examples](#code-examples)

- [Reference linking with @see tag](#reference-linking-with-see-tag)

- [Documenting Hooks](#documenting-hooks)

<!-- /TOC -->

## Pull Requests

Pull requests are highly appreciated. More than [150 people](https://github.com/timber/timber/graphs/contributors) have written parts of Timber (so far). Here are some guidelines to help:

1. \*\*Solve a problem\*\* – Features are great, but even better is cleaning-up and fixing issues in the code that you discover.

2. \*\*Write tests\*\* – This helps preserve functionality as the codebase grows and demonstrates how your change affects the code.

3. \*\*Write documentation\*\* – Timber is only useful if its features are documented. This covers inline documentation of the code as well as documenting functionality and use cases in the Guides section of the documentation.

4. \*\*Small > big\*\* – Better to have a few small pull requests that address specific parts of the code, than one big pull request that jumps all over.

5. \*\*Comply with Coding Standards\*\* – See next section.

## Coding Standards

We try to follow the [WordPress Coding Standards](https://make.wordpress.org/core/handbook/coding-standards/php/) as close as we can, with the following exceptions:

- Class and file names are defined in `StudlyCaps`. We follow the [PSR-0 standard](http://www.php-fig.org/psr/psr-1/#namespace-and-class-names), because we use autoloading via Composer.

- We use hook names namespaced by `/` instead of underscores (e.g. `timber/context` instead of `timber\_context`).

### Use PHP\_CodeSniffer to detect coding standard violations

To check where the code deviates from the standards, you can use [PHP\_CodeSniffer](https://github.com/squizlabs/PHP\_CodeSniffer). Timber comes with a `phpcs.xml` in the root folder of the repository, so that the the Timber code base will be automatically checked for coding standards violations.

#### Command Line Usage

When you run `composer install` in Timber’s repository root, you will get all required dependencies to check the coding standards.

To run PHP\_CodeSniffer with the default settings on all relevant Timber files, use the following command from the root folder of the Timber repository:

```bash

composer lint

```

You can check a single file like this:

```bash

./vendor/bin/phpcs ./lib/Menu.php

```

Use `./vendor/bin/phpcs --help` for a list of available settings or refer to the [PHP\_CodeSniffer documentation](https://github.com/squizlabs/PHP\_CodeSniffer/wiki).

#### Use it in your IDE

Please refer to <https://github.com/WordPress-Coding-Standards/WordPress-Coding-Standards#using-phpcs-and-wpcs-from-within-your-ide> for different ways to use PHP\_CodeSniffer directly in your IDE. In some IDEs like PHPStorm, you may have to select the `phpcs.xml` explicitly to apply the proper standards.

#### Whitelisting

If it’s not possible to adapt to certain rules, code could be whitelisted. However, this should be used sparingly.

- <https://github.com/WordPress-Coding-Standards/WordPress-Coding-Standards/wiki/Whitelisting-code-which-flags-errors>

- <https://github.com/squizlabs/PHP\_CodeSniffer/wiki/Advanced-Usage#ignoring-parts-of-a-file>

## Inline Documentation

The [Reference section](https://timber.github.io/docs/reference/) of the documentation is automatically generated from the inline documentation of the Timber code base. To document Timber, we follow the official [PHP Documentation Standards](https://make.wordpress.org/core/handbook/best-practices/inline-documentation-standards/) of WordPress.

There are minor differences to the official standards:

- For class headers, we don’t use `@package` or `@subpackage` tags.

- We don’t use the `@access` tag.

### Ignoring Structural Elements

The documentation generator ignores a number of elements. An element (class, method, property) is \*\*ignored when one of the following conditions\*\* applies:

- No DocBlock is provided

- No `@api` tag is present

- An `@ignore` tag is present

- An `@internal` tag is present

- The visibility is `private` (applies to methods only)

This means that for Markdown files to be generated for a class at all, you’ll need at least a DocBlock with an `@api` tag.

### Referencing class names

When referencing a namespaced class name in a type (for example in a `@param` or `@return` tag), then use the fully qualified name. Example: `Timber\Post` instead of just `Post`.

### Code examples

Timber uses tabs for indentation, but you should always use spaces for indentation in code examples, because the resulting Markdown will have a more consistens styling.

The `@example` tag allows you add code examples to your DocBlocks, including fenced code blocks:

```php

/\*\*

\* Function summary.

\*

\* Function description.

\*

\* @api

\* @example

\*

\* Optional text to describe the example.

\*

\* ```php

\* my\_method( 'example', false );

\* ```

\*

\* @param string $param1 Description. Default 'value'.

\* @param bool $param2 Optional. Description. Default true.

\*/

function my\_method( $param1, $param2 = true ) {}

```

### Reference linking with @see tag

When you use the `@see` tag, the Reference Generator will automatically convert it to a link to the [reference](http://timber-docs.test/docs/reference/).

- Use this tag only when the referenced method has an `@api` tag, which means that it is public.

- Beware, you’ll always use the notation with `::`, which you normally know from static methods. But even if the method that you link is not static, you’ll have to use the double colon.

An example:

```

@see \Timber\Image::src()

```

will turn into this:

```html

<strong>see</strong>

<a href="/docs/reference/timber-image/#src">Timber\Image::src()</a>

```

### Documenting Hooks

Follow the official [Inline Documentation Standards for Actions and Filters](https://make.wordpress.org/core/handbook/best-practices/inline-documentation-standards/php/#4-hooks-actions-and-filters) to document hooks.

#### Keywords for filters and actions

A DocBlock that documents a hook should always start with the keyword

- `Filters` for filter hooks

- `Fires` for action hooks

This is needed so that the [Hook Reference Generator](https://github.com/timber/teak/blob/master/lib/Compiler/HookReference.php) can detect the comments associated with filters and actions.

#### Syntax

```php

/\*\*

\* Filters … / Fires … (Summary)

\*

\* Description.

\*

\* `$var` Optional description of variables that might be used in the filter name.

\*

\* @see \Timber\Classname::function()

\* @link https://github.com/timber/timber/pull/1254

\* @since x.x.x

\* @deprecated x.x.x

\* @example

\* Optional text to describe the example.

\*

\* ```php

\* // A PHP example

\*

\* /\*\*

\* \* Multiline comments are possible as well.

\* \* You’ll need to escape the closing tag with a "\".

\* \*\/

\* ```

\*

\* ```twig

\* {# A Twig example #}

\* ```

\*

\* @param type $var Description. Default 'value'.

\* @param array $args {

\* Short description about this hash.

\*

\* @type type $var Optional. Description. Default value.

\* @type type $var Optional. Description. Default value.

\* }

\* @param type $var Optional. Description. Default value.

\*/

```

#### Dynamic filters

When a filter contains a variable, it should be marked up with double quotes `"` and the variable name inside curly braces:

```php

$force = apply\_filters( "timber/transient/force\_transient\_{$slug}", $force );

```

Additionally to this, document what the variable is by adding it to the description. Add it on a new line with the variable wrapped in backticks, so that it appears as code in Markdown:

```php

/\*\*

\* Filters the status of a particularly named transient.

\*

\* Allows you to override the behavior on a case-by-case basis.

\* `$slug` The transient slug.

\*

\* @param bool $force Param description.

\*/

$force = apply\_filters( "timber/transient/force\_transient\_{$slug}", $force );

```

#### Multiline declaration

Formatting a filter into multiple lines when the line should be too long is allowed:

```php

/\*\*

\* Filters …

\*/

$force = apply\_filters\_deprecated(

'timber\_force\_transients',

array( $force ),

'2.0.0',

'timber/transient/force\_transients'

);

```

#### Unfinished filters

If a filter description is not finished yet, mark it up with the `@todo` tag. It’s okay if you don’t know what a filter is doing exactly or if you’re unsure about what a parameter does. Describe what needs to be done in the `@todo` tag.

```php

/\*\*

\* Filters …

\*

\* @todo Add summary, add description.

\*

\* @param bool $force Param description.

\*/

```

As soon as the todo is resolved, the `@todo` tag can be removed.

## Process

All PRs receive a review from at least one maintainer. We’ll do our best to do that review in a week, but we’d rather go slow and get it right than merge in code with issues that just lead to trouble.

### GitHub reviews & assignments

You might see us assign multiple reviewers, in this case these are OR checks (i.e. either Coby or Pascal) unless we explicitly say it’s an AND type thing (i.e. can both Lukas and Maciej check this out?).

We use the assignee to show who’s responsible at that moment. We’ll assign back to the submitter if we need additional info/code/response, or it might be assigned to a branch maintainer if it needs more thought/revision (perhaps it’s directly related to an issue that's actively being worked on).

Once approved, the lead maintainer for the branch should merge the PR into the `master` or `2.x` branch. The 1.x team will work to resolve merge conflicts on #1617 (`2.x` into `master`) so the branches stay in sync.

### Branch Maintainers

\* 1.x: @jaredNova (lead), @palmiak

\* 2.x: @gchtr (lead), @pascalknecht, @cobytamayo