

Advanced Custom Fields



- [Getting data from ACF](#)
 - [Transform values to Timber/PHP objects](#)
 - [Unformatted values](#)
- [WYSIWYG Field \(and other requiring text\)](#)
- [Image Field](#)
 - [The quick way \(for most situations\)](#)
 - [The long way \(for some special situations\)](#)
- [Gallery Field](#)
- [Group Field](#)
- [Relationship field](#)
- [Repeater Field](#)
 - [Nested Repeater fields](#)
 - [Troubleshooting Repeaters](#)
- [Flexible Content Field](#)
 - [Repeater in Flexible Content Field](#)
 - [Split Flexible Content Fields into includes / chunks](#)
- [Options Page](#)
 - [Get all info from your options page](#)
 - [Use options info site wide](#)
- [Getting ACF info](#)

Timber is designed to play nicely with the amazing [Advanced Custom Fields](#) plugin.

Getting data from ACF

If you've worked with ACF before, you're use to use `get_field( 'my_acf_field' )` all the time. In Timber, getting data from ACF works the same way as getting WordPress's native meta data in general:

Twig

```
{{ post.meta('my_acf_field') }}
```

PHP

```
$meta = $post->meta('my_acf_field');
```

Transform values to Timber/PHP objects

Timber by default returns all field values as is based on the return type set in your ACF field setting.

But sometimes you might want to transform values directly into Timber/PHP objects. For example, if you have a relationship field, you might want to transform the values directly into `Timber\Post` objects.

You can do so using the `timber/meta/transform_value` filter:

functions.php

```
add_filter('timber/meta/transform_value', '__return_true');
```

Or you can use the `transform_value` parameter to transform values on a field by field basis:

Twig

```
{{ post.meta('my_acf_field', { transform_value: true }) }}
```

PHP

```
$meta = $post->meta('my_meta_field', ['transform_value' => true]);
```

You can also use both the filter and parameter options at the same time to globally transform values and then opt-out on a field by field basis by setting `transform_value` to `false`.

The values of the following field types will be transformed into Timber/PHP objects when using transforms:

Field type	Returns
File	<code>Timber\Attachment</code>
Image	<code>Timber\Image</code>

Field type	Returns
Gallery	array of <code>Timber\Post</code> objects
Date picker	<code>DateTimeImmutable</code>
Date time picker	<code>DateTimeImmutable</code>
Post object	array of <code>Timber\Post</code> objects
Relationship	array of <code>Timber\Post</code> objects
Taxonomy	array of <code>Timber\Term</code> objects
User	array of <code>Timber\User</code> objects

Unformatted values

In ACF, all values are filtered. If you want to use unfiltered, raw values from the database, you're probably used to using the third parameter for `get_field()`, which is called `$format_value`. This defaults to true. In Timber, you'd pass it like so:

Twig

```
{{ post.meta('my_acf_field', { format_value: false }) }}
```

PHP

```
$meta = $post->meta('my_acf_field', [
    'format_value' => false,
]);
```

You can also use the **faster** `raw_meta()` **method**, which accesses values directly from the database and bypasses any ACF filters:

Twig

```
{{ post.raw_meta('my_acf_field') }}
```

WYSIWYG Field (and other requiring text)

```
<h3>{{ post.title }}</h3>

<div class="intro-text">
    {{ post.meta('my_wysiwyg_field') }}
</div>
```

This will apply your expected paragraph breaks and other pre-processing to the text.

Image Field

You can retrieve an image from a custom field, then use it in a Twig template. The most reliable approach is this: When setting up your custom fields, you'll want to use `Image ID` as the field's return value. The image object, URL, etc. *should* work, but it's not as fool-proof.

The quick way (for most situations)

```

```

The long way (for some special situations)

This is where we'll start in PHP.

single.php

```
$post = Timber::get_post();

$context = Timber::context([
    'post' => $post,
]);

Timber::render('single.twig', $context);
```

`Timber\Image` should be initialized using a WordPress image ID. It can also take URLs and image objects, but that requires extra processing.

You can now use all the above functions to transform your custom images in the same way, the format will be:

```
{% if post.meta('hero_image') %}
    
{% endif %}
```

Gallery Field

```
{% for image in post.meta('gallery') %}
    
{% endfor %}
```

Group Field

```
{{ post.meta('group').first_field }}
{{ post.meta('group').second_field }}
```

or

```
{% set group = post.meta('group') %}
{{ group.first_field }}
{{ group.second_field }}
```

Relationship field

The post data returned from a relationship field will not contain the Timber methods needed for easy handling inside of your Twig file. To get these, you'll need to convert them into proper

`Timber\Post` objects using `get_posts()`:

```
{% for item in get_posts(post.meta('relationship_field')) %}
    {{ item.title }}
    {# Do something with item #}
{% endfor %}
```

Repeater Field

You can access repeater fields within Twig files:

single.twig

```
<h2>{{ post.title }}</h2>
<div class="my-list">
    {% for item in post.meta('my_repeater') %}
        <div class="item">
            <h4>{{ item.name }}</h4>
            <h6>{{ item.info }}</h6>
            
        </div>
    {% endfor %}
</div>
```

Nested Repeater fields

When you run `meta` on an outer ACF field, everything inside is ready to be traversed. You can refer to nested fields via `item_outer.inner_repeater`

```
{% for item_outer in post.meta('outer') %}
    {{ item_outer.title }}

    {% for item_inner in item_outer.inner_repeater %}
        {{ item_inner.title }}
    {% endfor %}

{% endfor %}
```

Troubleshooting Repeaters

A common problem in working with repeaters is that you should only call the `meta` method **once** on an item. In other words if you have a field inside a field (for example, a relationship inside a repeater or a repeater inside a repeater, **do not** call `meta` on the inner field). More:

DON'T DO THIS: (Bad)

```
{% for gear in post.meta('gear_items') %}
    <h3> {{ gear.brand_name }} </h3>
    {% for gear_feature in gear.meta('features') %}
```

```
        <li> {{ gear_feature }} </li>
    {% endfor %}
{% endfor %}
```

DO THIS: (Good)

```
{% for gear in post.meta('gear_items') %}
    <h3> {{ gear.brand_name }} </h3>
    {% for gear_feature in gear.features %}
        <li> {{ gear_feature }} </li>
    {% endfor %}
{% endfor %}
```

Flexible Content Field

Similar to repeaters, get the field by the name of the flexible content field:

```
{% for media_item in post.meta('media_set') %}
    {% if media_item.acf_fc_layout == 'image_set' %}
        
        <p class="caption">{{ get_image(media_item.image).caption }}</p>
        <aside class="notes">{{ media_item.notes }}</aside>
    {% elseif media_item.acf_fc_layout == 'video_set' %}
        <iframe width="560" height="315"
src="https://www.youtube.com/embed/{{media_item.youtube_id}}" frameborder="0"
allowfullscreen></iframe>
        <p class="caption">{{ media_item.caption }}</p>
    {% endif %}
{% endfor %}
```

Repeater in Flexible Content Field

Similar to nested repeaters, you should only call the `meta` method once when you use a repeater field inside a flexible content field:

```
{% for media_item in post.meta('media_set') %}
    {% if media_item.acf_fc_layout == 'image_set' %}
        {% for image_item in media_item.image_set %}
```

```

        
        <p class="caption">{{ get_image(image_item.image).caption }}</p>
        <aside class="notes">{{ image_item.notes }}</aside>
    {% endfor %}
{% endif %}
{% endfor %}

```

Split Flexible Content Fields into includes / chunks

We can break Flexible Content Fields into small blocks with include files utilizing the `acf_fc_layout` value. This way you have more flexibility and reusability for included sections. For instance, this is a great way to build landing pages that re-use the same blocks in different configurations.

You could use a **blocks** subdirectory where you put all your Twig template files for your blocks. For example: **wp-content/themes/example-theme/views/blocks**.

```

{% for block in post.meta( 'blocks' ) %}
    {{ include("blocks/#{ block.acf_fc_layout|sanitize } .twig", {
        block: block
    }) }}
{% endfor %}

```

The [sanitize](#) filter will slugify the block name.

Consider this example for a Flexible Content Field named `Photo` with a text field named `credit`. You would create a **blocks/photo.twig** file which is automatically included:

```

<p>Photo by: {{ block.credit }}</p>

```

To prevent errors with include files that can't be found, you can optionally use the `ignore_missing` parameter for `include()`:

```

{% for block in post.meta( 'blocks' ) %}
    {{ include("blocks/#{ block.acf_fc_layout|sanitize } .twig", {
        block: block
    }, ignore_missing = true) }}
{% endfor %}

```

Options Page

PHP

```
$context['site_copyright_info'] = get_field('copyright_info', 'options');

Timber::render('index.twig', $context);
```

Twig

```
<footer>{{ site_copyright_info }}</footer>
```

Get all info from your options page

```
$context['options'] = get_fields('options');

Timber::render('index.twig', $context);
```

ACF Pro has a built in options page, and changes the `get_fields( 'options' )` to `get_fields( 'option' )`.

```
<footer>{{ options.copyright_info }}</footer>
```

Use options info site wide

To use any options fields site wide, add the `option` context to your **functions.php** file:

```
add_filter('timber/context', 'global_timber_context');

/**
 * Filters global context.
 *
 * @param array $context An array of existing context variables.
 * @return mixed
 */
function global_timber_context($context)
{
    $context['options'] = get_fields('option');
```

```
return $context;
}
```

Now, you can use any of the option fields across the site instead of per template.

footer.twig

```
<footer>{{ options.copyright_info }}</footer>
```

Getting ACF info

You can grab specific field label data like so:

single.php

```
$context['acf'] = get_field_objects($data['post']->ID);
```

```
{{ acf.your_field_name_here.label }}
```

📄 #2

📄 📄 📄 2026-08-29 16:14:01 UTC

📄 📄 📄 2026-08-29 16:14:01 UTC