Mathematical and Software Engineering, Vol. 8, No. 1-2 (2022), 1-9.
Varɛpsilon Ltd, http://varepsilon.com
DOI 10.5281/zenodo.7091882

Comparison of template engines of PHP frameworks

Veselina Spasova[1] **and Oleksii Raiu**[1]

[1]Department of Computer Science, Varna Free University Chernorizets Hrabar, Bulgaria

Abstract

In this study, we compared two common template Engines - Blade and Twig, which are closely related, but have some very strong differences. Blade belongs to the Laravel CMF, which is built on top of the Symfony CMF. The Symfony CMF itself has its own template engine called Twig. As a result, we want to find answers of several important questions: what are the differences between Blade and Twig that made it desirable for the Laravel developers to come up with a new template engine besides Twig; how do Twig and Blade answer to developer needs and to make some objective suggestions that can help choosing between the two engines.

Keywords: PHP, Symfony, Laravel, performance, tools comparison

1 Introduction

According to W3CTechs, PHP holds 77.6% of the server-side languages market share [1]. PHP was first released on 1995, and has since gained popularity. It has had many rivals, each claiming technological supremacy, but even 3 decades later; it still holds an undisputed lead in market share, with the leading CMS (Content Management System) and CMF (Content Management Framework) using PHP. Over the years, PHP has undergone significant changes to its architecture, and between 5 and 7 (the jump skipped version 6 completely), it was significantly rewritten and redesigned for better performance and OOP support.

Being the most widely used server-side language, one would expect for it to have a powerful and robust templating system that would be used to integrate the logic and the markup. PHP originally developed as a “Hypertext Preprocessor”, and was meant to be inserted into the markup language (HTML) as snippets of code. Originally, that served the idea very well, and even as of 2022, there is no template engine in CMS giants like WordPress. Many PHP based websites today still use raw PHP to embed logic in markup, WordPress not being an exception.

Joomla, the second most popular CMS after WordPress, also uses raw PHP for templating.

Most modern PHP template engines are PHP-based and are compiled and cached on the server for faster execution. The initial template is a text file, written using the template engine's language. On page execution, the template engine checks the files whether they have been updated, and then compiles those template files into PHP and caches them on the server. Sometimes the compiled templates are cached in RAM, but more often they are saved in a protected directory as PHP files. If a rendering engine finds a compiled template PHP file, it will include that PHP file, and pass the available variables to it. If the compiled template is unavailable, it will call the template engine to find and compile the template. When a template has been updated, its cache file will be flushed, causing the template to be recompiled.

In this study, we will compare two common template Engines - Blade and Twig, which are closely related, but have some very strong differences. Blade belongs to the Laravel CMF, which is built on top of the Symfony CMF. The Symfony CMF itself has its own template engine called Twig.

The first criticism is that raw PHP in markup breaks the principle of separation of business logic and presentation. It is a type of a SoC (Separation of Concerns) Software Engineering principle that states that different logical parts of application should not be mixed [2]. Mixing of business logic and presentation results in code that is hard to read and support, and is prone for bugs. Templating allows to abstract the presentation.

Another benefit of using templates and keeping the SoC principle is organizational. Template engines are simpler and more intuitive than the raw PHP, and can be mastered without having to know the depths of a given platform. Thus, an effective Back End (BE) and Front End (FE) separation can be achieved, with FE developers receiving the variables that they can use in their templates, and then working abstracted from the BE using the provided variables and the template engine.

The second criticism for using raw PHP is that when used in the presentation layer, it becomes increasingly less secure. One reason is that the presentation layer deals with the output of user input, exposing the scenarios where maliciously submitted code can become published on a web page. Another reason is human factor - it's usually front-end developers who work with the presentation layer, and front end developers are generally, due to organizational separation of concerns, are much weaker in writing secure code.

Using template engines allows back end developers to provide tools for "sandboxed" interactions, where the output variables are properly sanitized, and front-end developers have access to the presentation related template logic without having to make security decisions themselves - all security decisions are made by the back-end developers who provide the variables for them.

It is customary for the PHP template engines to be provided as libraries that can be embedded into any PSR-compatible PHP web application. Some,

however, rely on a specific framework that they are a part of. With the libraries that we will use, Twig is reusable, but Blade relies on the Laravel framework.

Twig and Blade belong to Symfony and Laravel frameworks. Laravel is built on top of Symfony and preserves its MVC architecture. Thus, both template engines are used for the View implementation, and the SoC is built into them both.

There are more than a dozen PHP template engines overall, however, there are only a few very popular ones. According to Wappalyzer, one of the leading web technology analytical tools, the most popular PHP CMF is Laravel [3]. Also, according to Wappalyzer, the most popular CMS that uses a template engine is Drupal [4]. Drupal is built on Symfony and uses the Twig template engine. Thus, we can safely say that Twig and Blade are the most used PHP template engines nowadays. These will be the two engines that we will compare in this study.

Another interesting aspect of these two engines is that both Twig and Blade can be used in Laravel. Blade is Laravel's native template engine, while Twig can be installed in Laravel and used as a replacement for Blade for templating.

From the above, it is very clear that comparing Twig and Blade is not only reasonable for their popularity, but also because they both can be attached to the same application, thus eliminating the difference that an underlying framework (CMF) can bring into performance testing.

Twig and Blade are both template engines, but their implementation is very different. Twig is a separate library that can be installed, auto loaded, and used for any third-party PHP application. Composer, the standard PHP library manager, is the standard tool to install the “twig/twig” library into the project, and make it autoloaded. To make Twig work with Laravel, there is a bridge available [5]. After a simple settings adjustment, the bridge enables the use of the Twig template engine inside Laravel.

Blade template engine is embedded in Laravel as a part of the “\Illuminate\” namespace, which is the standard namespace for the whole framework. Thus, Blade is tightly coupled with the rest of Laravel, and cannot be easily abstracted for reuse.

Both Twig and Blade will compile the templates into raw PHP files and cache them until that cache is cleared.

Twig and Blade have their own syntax, and they play the same role - allowing passing variables to markup and having some basic presentation-related logic, making sure that the variables are properly escaped, without exposing the whole of the back end to the front-end developer. This gives the front-end developer the freedom they need, but at the same time, it minimizes the risks of having unescaped variables displayed, and disincentives the front-end developer from including business logic and mixing it with the markup, keeping the templates simple and readable, and minimizing the amount of introduced bugs.

2 Results and Discussion

Modern template engines have common functionality which is expected of them in order to be effective. This usually includes functionality for: handling variables; security and output escaping; rendering logic; handling of components and plugins. We will make the assessment from the perspective of the application developer and the requirements (needs, expectations) regarding the tools used. We will successively test the features of both frameworks.

2.1. Handle and print simple and complex (Variables handling)

Blade

```
{{-- COMMENT: Blade does not support assignment, uses raw
@php($name = "John")
Hello, {{ $name }}
@php($user = ['name' => $name])
Array name: {{ $user['name'] }}
@php($user = new stdClass())
@php($user->name = $name)
Object name: {{ $user->name }}
```

Twig

```
{% set name = 'John' %}
Hello, {{ name }}
{% set user = {'name': name} %}
Array name: {{ user.name }}
Array dot notation name: {{ user.name }}

{# Twig can print objects, but not create objects. #}
{# Array printed twice to for performance adjustment. #}
```

Figure 1. Blade and Twig variables handling code

Twig has built-in functionality to declare variables, while Blade makes the user resort to the raw PHP directive for doing it. Related to ability to assign value to variables regardless of type, including to arrays and public objects classes - Blade does not handle this functionality, delegating it to raw PHP, Twig can handle this functionality inside the sandbox. Both Blade and Twig performed well, providing the ability to access array keys and object values within the print syntax.

2.2. Need to apply logic to parts of templates (Use logic)

Blade

```
@php($test = 5)

{{-- If conditional. --}}
@if ($test === 5)
  Test has value of 5.
@elseif ($test < 5)
  Test is smaller than 5.
@else
  Test is greater than 5.
@endif

{{-- Switch statement. --}}
@switch($test)
  @case($test === 5)
  Test has value of 5.
  @break
  @case($test < 5)
  Test is smaller than 5.
  @break
  @default
  Test is greater than 5.
@endswitch
```

Twig

```
{% set test = 5 %}

{# If conditional. #}
{% if test == 5 %}
  Test has value of 5.
{% elseif test < 5 %}
  Test is smaller than 5.
{% else %}
  Test is greater than 5.
{% endif %}

{# Symfony does not support "switch". #}
{# An "if" implementation has been placed for performance
{% if test == 5 %}
  Test has value of 5.
{% elseif test < 5 %}
  Test is smaller than 5.
{% else %}
  Test is greater than 5.
{% endif %}
```

Figure 2. Blade and Twig rendering logic code

Rendering logic decides if a certain part of template is getting rendered, or

renders a part of template out of a set of parts. Example would be a login section that displays a username linked to the user account when the user is logged in, and a link to a login form or an embedded login form, if the user is not logged in. One of the multiple rendering choices would be rendering a file record in a certain way, depending on whether that file is a document, a video, an audio, or an image.

Twig, does not support "switch". Twig's usage of "ifs" is much shorter vertically and is much more compact that Blade's "switch". Switch operator can be longer vertically, but it's structured to allow the developer to better discern the multiple conditions in code, and thus, it improves maintainability and readability of code.

2.3. Need to have iterations and loops (Iterators, Loops)

Blade

```
@php($names = ['John', 'Linda', 'Thomas', 'Elsa'])
@php($count = count($names))

{{-- Foreach. --}}
<ol>
@foreach ($names as $name)
 <li>{{ $name }}</li>
@endforeach
</ol>

{{-- Loop. --}}
<ol>
@for ($i = 0; $i < $count; $i++)
    <li>{{ $names[$i] }}</li>
@endfor
</ol>
```

Twig

```
{% set names = ['John', 'Linda', 'Thomas', 'Elsa'] %}
{% set count = names|length - 1 %}

{# Foreach. #}
<ol>
{% for name in names %}
  <li>{{ name }}</li>
{% endfor %}
</ol>

{# Loop. Looks somewhat stretched. #}
<ol>
{% for i in 0..count %}
  <li>{{ names[i] }}</li>
{% endfor %}
</ol>
```

Figure 3. Blade and Twig control structures code

We need to have a "for … next" control structure that will iterate the loop over a given N number of times. Blade has this syntax, but Twig, which has already misleadingly implemented "for … in" as an iterator, simply does not have this functionality.

2.4. Need to have template inclusion and inheritance (Inclusion, Inheritance)

Blade

```
@extends('blade.includes.inherited')

@section('content')
@parent
<p>This text belongs to the child override section.</p>
@endsection
```

Twig

```
{% extends 'includes/inherited.html.twig' %}

{% block content %}
{{ parent() }}
<p>This text belongs to the child override section.</p>
{% endblock %}
```

Figure 4. Blade and Twig template inheritance code

Developers want to be able to both include reusable parts of templates (horizontal inheritance), and to include lower level templates in my higher-level templates (vertical inheritance). Not only the templates need to be included, but they also should handle variables in a smart way, so that the parent templates could

pass variables to child templates.

Both template engines can include reusable components, this feature is clearly implemented in both template engines in a way that resamples the “include” syntax from PHP itself. Both template engines follow the pattern of defining “blocks” that the higher-level templates can override, and providing the vertical inheritance through the “extends” syntax, using the OOP class inheritance syntax to indicate that one template extends another.

2.5. Need the templates to be secure (Security)

Blade

```
{{-- 1. Prevents use of raw PHP. NO. --}}
@php($response = app()->version())
App version: {{ $response }}

{{-- 2. Prevents writing to model. Yes but doable through
{{-- 3. Prevents use of direct MySQL. No but doable throu
```

Twig

```
{# 1. Prevents use of raw PHP. YES. #}
{# 2. Prevents writing to model. YES. #}
{# 3. Prevents use of direct MySQL. YES. #}
```

Figure 5. Blade and Twig security comparison code

Twig does not expose raw PHP. Blade not only exposes raw PHP to Front End developers, it actually requires them to use raw PHP for some very basic functionality, such as variable declaration. Twig prevents writing to model, and Blade does not, mainly by exposing PHP. Using PHP, the front-end developer can modify the state of the application or website, both within the request and also permanently. Twig does not allow running MySQL commands, but Blade allows it, by exposing access to raw PHP.

2.6. Need the templates to be stable (Stability)

Blade

```
{{-- Assigning or printing an empty or undeclared variabl
@@{{ $a }} {{-- ERROR: "$a is undefined" --}}

{{-- Assigning or printing an empty or undeclared array k
@php($a = [])
@@{{ $a[0] }} {{-- ERROR: "Undefined offset: 0" --}}

{{-- Operations on variables of different types. --}}
@php($a = 1)
@php($b = '1')
{{ $a + $b }}

{{-- Empty and undeclared variables and keys in logic and
@@if ($test === 5) {{-- ERROR: "$test is undefined" --}}
@@endif

{{-- Variables of wrong types in logic and control struct
@php($a = 1)
@php($b = TRUE)
@if ($a + $b == 2)
<p>TRUE</p>
@endif
```

Twig

```
{# Assigning or printing an empty or undeclared variable.
{{ a }}

{# Assigning or printing an empty or undeclared array key
{% set a = [] %}
{{ a[0] }}

{# Operations on variables of different types. #}
{% set a = 1 %}
{% set b = '1' %}
{{ a + b }}

{# Empty and undeclared variables and keys in logic and c
{% if test == 5 %}
{% endif %}

{# Variables of wrong types in logic and control structur
{% set a = 1 %}
{% set b = TRUE %}
{% if (a + b == 2) %}
<p>TRUE</p>
{% endif %}
```

Figure 6. Blade and Twig stability test code

The first point is for keeping stability while assigning or printing an empty or undeclared variable. An empty variable, if printed, is easily nullable, and there is no reason to raise an error if "nothing" can be printed instead. Twig succeeds, but Blade fails the test, raising an error and breaking the normal output.

The second point goes one step further and checks for stability while assigning or printing an empty or undeclared array key and object property. Twig scores a point, while Blade rises an error "undefined offset", making the template unstable.

The third point is for checking operations with different variable types. PHP has dynamic data types, which means that you can assign any value to any variable irrespective of type. In our tests, both Twig and Blade used the standard PHP's approach of left-to-right evaluation, where the type of the first variable defines the expected type of the second variable. Both template engines performed successfully.

The fourth point is for keeping stability while having empty and undeclared variables and keys in logic and control structures. Twig again scores, while Blade fails. This is expected after the general failure with undefined variables.

Fifth point is for keeping stability while having variables of wrong types in logic and control structures. Here we are adding a numeric variable and a boolean variable. The same native PHP mechanism of dynamic casting applies, and the boolean is converted to its numeric value of 1. Both template engines pass the test.

2.7. Need to access relevant context data (Context)

Blade

```
{{-- Blade does not have global context, uses PHP to get
{{ app()->databasePath() }}
{{ app()->getLocale() }}
{{ app()->version() }}

{{-- Immediate context: YES (See Inheritance). --}}
```

Twig

```
{# Global context: YES #}
{{ dump(_context) }}
{{ dump(_self) }}
{{ dump(_charset) }}

{# Immediate context: YES (See Inheritance). #}
```

Figure 7. Blade and Twig context usage sample code

Both Twig and Blade ended up having some context variables, but of different types. Blade had very little context data, mainly because it exposes the raw PHP to the front-end developer. Twig exposes some relevant global context information, while Blade exposes the whole application object to draw from. Both engines can pass variables between templates when inheriting/extending templates.

2.8. Need to be able to extend the engine with plugins (Extend ability)

Blade

```
@custom_print('Blade can be extended.')
```

Twig

```
{{ custom_print('Twig can be extended.') }}
```

Figure 8. Blade and Twig custom function/directive usage code

Both template engines have functionality to extend themselves.

2.9. Need to keep compile/render times minimal (Performance)

	Blade	Twig
Time, total	136.69 ms	639.01 ms
Perf. index	d = 1	d = 0.2

Figure 9. Blade and Twig total performance check test result

In performance tests all templates are run in a loop of 100 iterations and performance index is calculated as deviation between winner time and looser time. Blade on the average is 5x times faster than Twig, which makes the performance index of Blade 1, and that of Twig 0.2. This performance is easily predictable given that Twig appears to be much more featuring rich and much “smarter” than Blade. But as always, there is a tradeoff between speed and amount of functionality.

3 Conclusion

After the application has been completed and run, we can evaluate the results, count the number of evaluation points, and review performance times. The total Table 1 of raw results from the application is listed.

On a quick look, it's obvious that Twig is much more feature-rich as a template engine, and can provide a rich feature set without making the front-end developer turn to raw PHP. At the same time, Twig is about 5 times slower than Blade. As we have already mentioned, this performance result will likely be leveled up by caching of the Twig template files. In fact, because Twig puts the main load on compiling and then renders very fast, we could expect that it would in fact outperform Blade when using precompiled templates (this, however, is out of scope of this study, even though it will certainly inform our final recommendations).

Table 1: Summary of measured competition points for Blade and Twig

Criteria	Blade	Twig
Variables handling	1	3
Rendering logic	2	1
Control structures	2	1
Template inheritance	2	2
Security	0	3
Stability	2	5
Context	2	2
Extend ability	1	1
Performance	1	0.2
TOTAL	**12 points, perf 1**	**18 points, perf 0.2**

References

[1] W3CTechs, Usage Statistics and Market Share of Server-side Programming Languages for Websites, March 2022, *W3Techs*, 2022, https://w3techs.com/technologies/overview/programming_language accessed March 2022.

[2] .NET Application Architecture Guide, 2nd Edition, Microsoft Press, 2009.

[3] Web frameworks market share, websites and contacts, *Wappalyzer*, https://www.wappalyzer.com/technologies/web-frameworks accessed March 2022.

[4] CMS market share, websites and contacts, *Wappalyzer*, https://www.wappalyzer.com/technologies/cms. Accessed March 2022.

[5] rcrowe/TwigBridge: Give the power of Twig to Laravel, *GitHub*, https://github.com/rcrowe/TwigBridge Accessed March 2022.