

DZone Events

Webinars

Evaluating Your Event Streaming Needs the Software Architect Way

Friday, March 31, 2023 | 1PM ET

SPONSORED BY



aven

intel

SAVE THE DATE!

Modern Type-Safe Template Engines (Part 2)

To wrap up this quick series on template engines, we do a feature and performance comparison of four interesting new solutions.

 by Miguel Gamboa ·  by Luis Duarte · Nov. 20, 18 · Tutorial

Like (4)

Comment (6)

Save

Tweet

in Share

15.39K Views

Join the DZone community and get the full member experience.

JOIN FOR FREE

DZone

DevOps

CI/CD, Application Delivery, and Release Orchestration

[New Report] DevOps: CI/CD, Application Delivery, and Release Orchestration

DZone's latest Trend Report explores how the nuanced derivatives of DevOps — IaC, shift left, source code management, CI/CD, and more — are continuing to change how we build, test, and deploy code today. [Get the report](#)

Welcome back! If you missed Part 1, you can check it out [here](#).

Feature Comparison

To give a better overview of these template engines, here's a table that provides a list of most important features:

FEATURE	ROCKER	J2HTML	KOTLINX HTML	HTMLFLOW
Functional Templates	x	✓	✓	✓
Elements validation	x	x	✓	✓
Attributes validation	x	x	x	✓
Full HTML API	✓	x	✓	✓
Well-Formed	x	✓	✓	✓
Performance	✓	x	x	✓

Most of these solutions tend to move the template definition from the textual files to functions defined in the environment language, i.e. Java or Kotlin. This removes the overhead of loading the textual files and parsing them at run-time. Another feature shared by all solutions, with Rocker as the exception, is the generation of well-formed documents.

One of the general problems presented at the beginning of this article and shared by most state of the art template engines is a lack of safety and that HTML is type checkless. KotlinX Html and HtmlFlow are the only engines that solve this issue. Yet, KotlinX Html is heavily handicapped when it comes to performance, being one of the worst in tested benchmarks, which will be presented in the next section.

Performance Evaluation

To perform an unbiased comparison we searched on GitHub and used the two most popular benchmarks for template engines. We integrated the missing engines in our forks of these benchmarks available at: [xmlet/template-benchmark](#) and [xmlet/spring-comparing-template-engines](#). These tests were done on a local machine with the following specs:

```
1 Microsoft Windows 10
2 Education OS Version: 10.0.17134
3 N/A Build 17134
4 Intel(R) Core(TM) i7-7700
5 HQ CPU @ 2.80GHz, 2808 Mhz, 4 Core(s), 8 Logical Processor(s) Java(TM) SE Runtime Environment 18.9 (build
```

Spring Benchmark

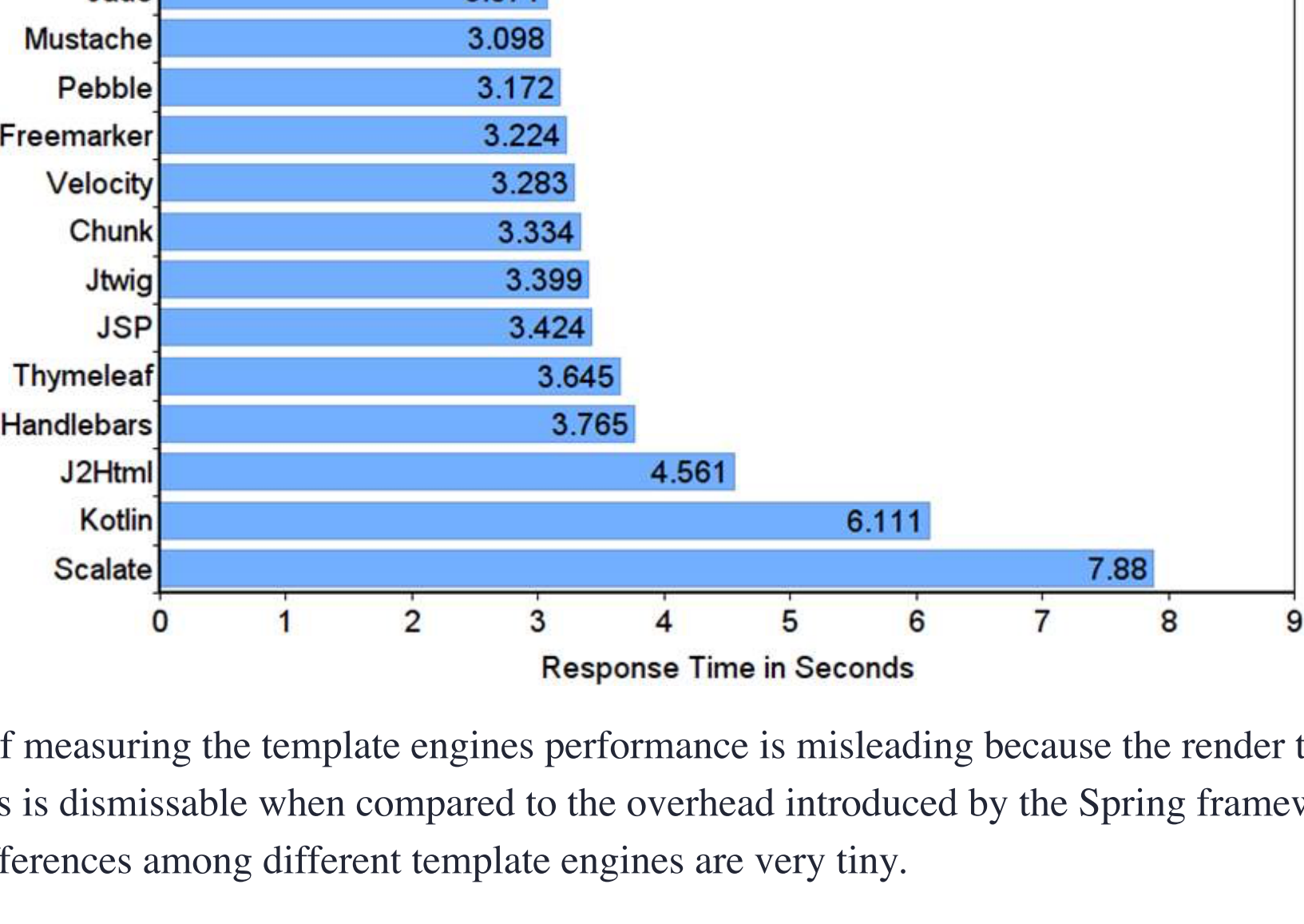
This benchmark uses the Spring 3 framework to host a web application that provides a route for each template engine to benchmark. Each template engine uses the same template (i.e. [Presentations](#)) and receives the same information to fill the template, which makes it possible to flood all the routes with a high number of requests and asserts which route responds faster, consequently asserting which template engine is faster. Following the benchmark recommendation we used the Apache HTTP server benchmarking tool as a stress tester running it with 25 concurrent requests and 25000 requests in total, which corresponds to the following settings:

```
1 ab -n 25000 -c 25 -k http://localhost:8080/<template engine>
```

The performance of each template engine was measured according to the guidelines specified in the [Spring benchmark repository](#), which requires:

at least two dry runs with the exact same settings, to make sure that initialization of the engines, warm up of the JVM and additional caches have taken place.

After that, we calculate the average time taken by 5 iterations of the same benchmark for each template engine. This resulted in the following numbers corresponding to the total time taken for processing 25000 requests with a concurrency level of 25 (**lower is better**).



This approach of measuring the template engines performance is misleading because the render time of the template engines is dismissable when compared to the overhead introduced by the Spring framework. Thus, the performance differences among different template engines are very tiny.

In this context, we mostly agree with the [template-benchmark](#) benchmark proposal, which uses JMh to implement the benchmark. So we end up using the [Presentations](#) template form the Spring benchmark and integrated it in our fork of [template-benchmark](#) to get more reliable measures. In the following listing, we show a short sample of the [Presentations](#) template in the Mustache idiom with only the dynamic parts:

```
1 <!DOCTYPE html>
2 <html>
3 ...
4 <body>
5   <div class="container">
6     ...
7     {{#presentationItems}}
8     <div class="panel panel-default">
9       <div class="panel-heading">
10        <h3 class="panel-title">
11          {{title}} - {{speakerName}}
12        </h3>
13      </div>
14      <div class="panel-body"> {{summary}} </div>
15    </div>
16  </body>
17 </html>
```

Template Benchmark

The advantage of this benchmark is that it focuses exclusively on evaluating the rendering process of each template engine. In this case, it does not use any web servers to handle a request, which is a more consistent approach. The general idea of this benchmark is the same, it includes many template engine solutions that define the same template and use the same data as a context object to generate the resulting HTML document.

But, in this case, it uses [Java Microbenchmark Harness](#), which is a Java tool to benchmark code. With JMh we indicate which methods to benchmark with annotations and configure different benchmark options such as the number of warm-up iterations, the number of measurement iterations, or the numbers of threads to run the benchmark method.

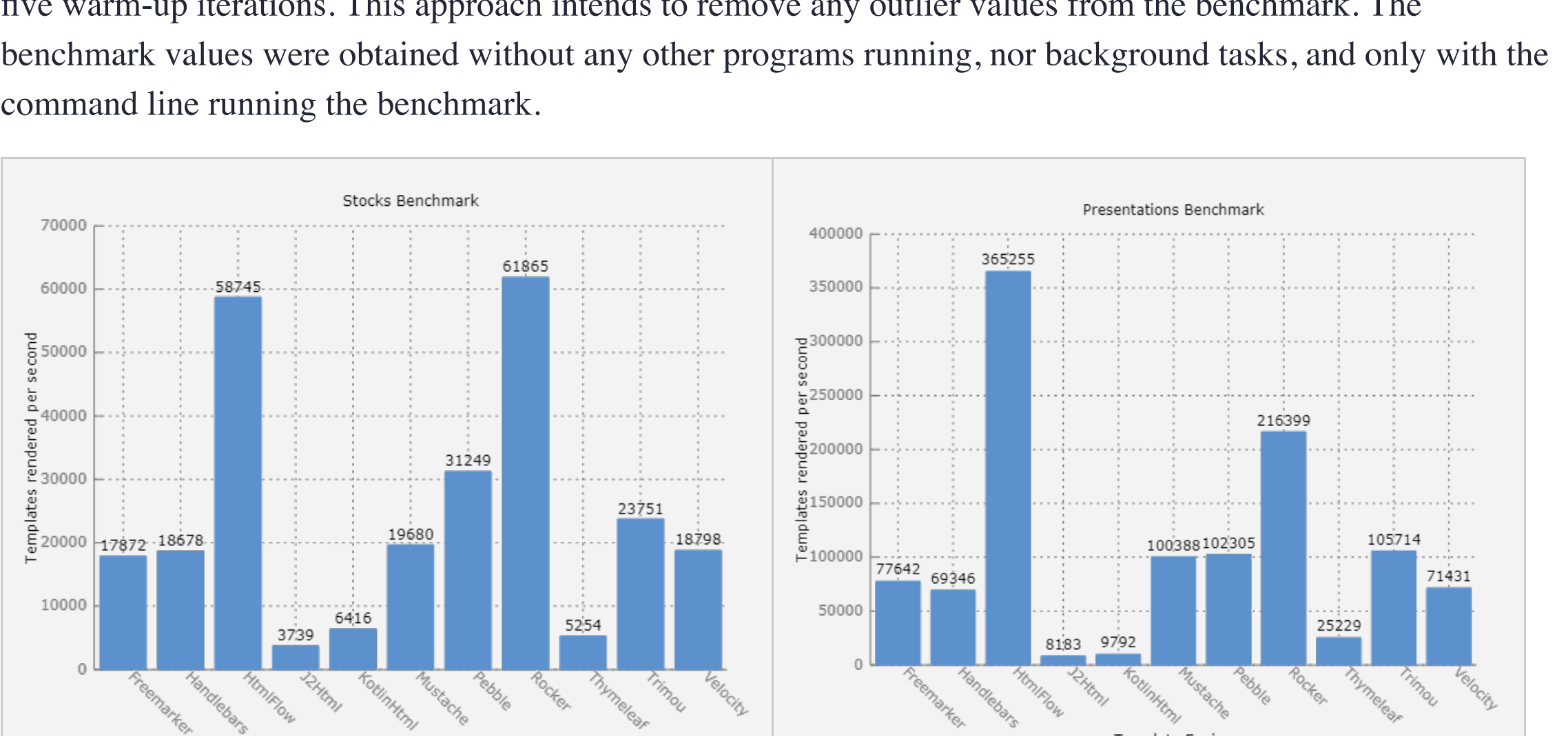
This benchmark contained eight different template engines: Freemarker, Handlebars, Mustache, Pebble, Thymeleaf, Trimou, Velocity, and Rocker. In addition, we integrated J2Html, KotlinX Html, and HtmlFlow.

The template-benchmark used only one template, i.e. [Stocks](#) template. In the following listing we show a short sample of the [Stocks](#) template in Mustache idiom with only the dynamic parts:

```
1 <!DOCTYPE html>
2 <html>
3 ...
4 <body>
5   ...
6   <table>
7     ...
8     <tbody>
9       {{#items}}
10      <tr class="{{rowClass}}">
11        <td>{{index}}</td>
12        <td><a href="/stocks/{{value.symbol}}">{{value.symbol}}</a></td>
13        <td><a href="{{value.url}}">{{value.name}}</a></td>
14        <td><strong>{{value.price}}</strong></td>
15        <td><strong>{{value.change}}</td>
16        <td><strong>{{value.ratio}}</td>
17      </tr> {{/items}}
18    </tbody>
19  </table>
20 </body>
21 </html>
```

By using two different templates the objective was to observe if the results were maintained throughout the different solutions. The main difference between both templates is that the [Stocks](#) template introduces much more binding operations: 1) it has more fields that will be accessed in the template, and 2) it has twenty objects in the default data set while [Presentations](#) only has ten objects in his data set. This means that the [Stocks](#) template will generate more String operations to the classic template engine solutions and more Java method calls for the solutions that have the template defined as a function in Java or Kotlin.

We ran the [template-benchmark](#) according to the requirements specified by its author. Thus the results are the mean value of five forked iterations, each one of the forks running eight different iterations, performed after five warm-up iterations. This approach intends to remove any outlier values from the benchmark. The benchmark values were obtained without any other programs running, nor background tasks, and only with the command line running the benchmark.



Regarding the classical template engines, i.e. Mustache, Pebble, Freemarker, Trimou, Velocity, Handlebars, and Thymeleaf, we can observe that most of them share the same level of performance, which should be expected since they all roughly have the same methodology. Regarding the remaining template engines, i.e. Rocker, J2Html, KotlinX Html, and HtmlFlow, the situation is diverse. On one hand, we have Rocker, which gives great performance when the number of placeholders increases, i.e. the [Stocks](#) benchmark, taking into consideration that it provides many compile-time verifications regarding the context objects, it presents a good improvement in comparison to the classical template engines. On the other hand, we have J2Html and KotlinX Html. Regarding J2Html we observe that the tradeoff of moving the template to a function of the environment language (i.e. Java) had a significant performance cost since it is consistently one of the two worst solutions performance-wise. Regarding KotlinX Html, its approach was definitely a step in the right direction since it validates the HTML rules and introduces compile-time validations, but, either due to the Kotlin language performance issues or poorly optimized code, it did not achieve the level of acceptable performance.

Lastly, HtmlFlow proved to be the best performance with the [Presentation](#) template. It achieved performance gains that surpass the second best solution by twice the operations per second. Regarding the [Stocks](#) template, the HtmlFlow held the top place even though the number of placeholders for dynamic information increased significantly. If we compare HtmlFlow to a similar solution, KotlinX, we observe a huge gain of performance on the HtmlFlow part.

In conclusion, HtmlFlow introduces domain language rule verification, removes the requirements of text files and additional syntaxes, adds many compile-time verifications, and, while doing all of that this, is still the best solution performance wise.

Conclusion

Despite the vast list of new template engines that appear every year, there are a few innovations in addition to a couple of new macros or idioms trying to enrich the set of available operations. However, all this is too restrictive in comparison to all modern features available in high-level programming languages such as Java 11 or Kotlin. DSL libraries for HTML such as [J2Html](#), [KotlinX.html](#), and [HtmlFlow](#) fill this gap, bringing Java or Kotlin to the template definition.

In this series, we presented four new technologies that break with the classical approach of textual template files and bring a new use for templates. These innovations introduced: HTML safety (in the case of [KotlinX.html](#) and [HtmlFlow](#)) and better performance (in the case of [Rocker](#) and [HtmlFlow](#)).

Final, the primary goal of DSL libraries for HTML is not, in truth, to compete with classic template engines, but to give alternatives ways of thinking on HTML dynamic build. So, we should not be only restricted with the classical proposal of textual template engines, which have poorly evolved in the last two decades, but also embrace innovative proposal such as DSL libraries.

Template

Engine

Opinions expressed by DZone contributors are their own.

Popular on DZone

- How We Solved an OOM Issue in TiDB with GOMEMLIMIT
- How To Build a Spring Boot GraalVM Image
- Master Spring Boot 3 With GraalVM Native Image
- Introduction to Container Orchestration

ABOUT US

Advertise with DZone

Let's be friends:

CONTRIBUTE ON DZONE

CONTACT US