MDPI

Article

Non-blocking Progressive Server-side Rendering (PSSR) Benchmark

Bernardo Pereira [1,*], Miguel Carvalho [2]

[1] Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa, 1950-062 Lisbon, Portugal
* Author to whom correspondence should be addressed.

Abstract: A single paragraph of about 200 words maximum. For research articles, abstracts should give a pertinent overview of the work. We strongly encourage authors to use the following style of structured abstracts, but without headings: (1) Background: place the question addressed in a broad context and highlight the purpose of the study; (2) Methods: describe briefly the main methods or treatments applied; (3) Results: summarize the article's main findings; (4) Conclusions: indicate the main conclusions or interpretations. The abstract should be an objective representation of the article, it must not contain results which are not presented and substantiated in the main text and should not exaggerate the main conclusions.

Keywords: keyword 1; keyword 2; keyword 3 (List three to ten pertinent keywords specific to the article; yet reasonably common within the subject discipline.)

Received:
Revised:
Accepted:
Published:

Citation: . Non-blocking Progressive Server-side Rendering (PSSR) Benchmark. *Computers* **2025**, *1*, 0. https://doi.org/

4. Benchmark Implementation

The benchmark is designed with a modular architecture, separating the *view* and *model* layers from the *controller* layer Model-View-Controller Pattern, which allows for easy extension and integration of new template engines and frameworks. It also includes a set of tests to ensure the correctness of implementations and to validate the *HTML* output.

The benchmark includes two different data models, defined as follows:

```
data class Presentation(
    val id: Long,
    val title: String,
    val speakerName: String,
    val summary: String
)

data class Stock(
    val name: String,
    val name2: String,
    val url: String,
    val symbol: String,
    val price: Double,
    val change: Double,
    val ratio: Double
)
```

Listing 1: Data Models

The application's repository contains a list of 10 instances of the `Presentation` class and 20 instances of the `Stock` class. Each list is used to generate a respective HTML view. Although the instances are kept in memory, the repository uses the `Observable` class from the *RxJava* library to interleave list items with a delay of 1 millisecond. This delay promotes context switching and frees up the calling thread to handle other requests in non-blocking scenarios, mimicking actual I/O operations.

By using the `blockingIterable` method of the `Observable` class, we provide a blocking interface for template engines that do not support asynchronous data models, while still simulating the asynchronous nature of the data source to enable PSSR. Template engines that do not support non-blocking I/O for PSSR include KotlinX, Rocker, JStachio, Pebble, Freemarker, Trimou, and Velocity. HtmlFlow supports non-blocking I/O through suspendable templates and asynchronous rendering, while Thymeleaf enables it using the `ReactiveDataDriverContextVariable` in conjunction with a non-blocking Spring `ViewResolver`.

The Spring WebFlux implementation uses Project Reactor to support a reactive programming model: each method returns a `Flux<String>` as the response body, which acts as a publisher that progressively streams the HTML content to the client. It also includes methods using Kotlin coroutines and other asynchronous mechanisms supported by template engines, such as the `writeAsync` method from HtmlFlow, which enables non-blocking I/O using *continuations*. The aforementioned blocking template engines are used in the context of Virtual Threads or alternative coroutine dispatchers, allowing the handler thread to be released and reused for other requests.

The Spring MVC implementation uses handlers based solely on the blocking interface of the `Observable` class. To enable PSSR in this context, we utilize the `StreamingResponseBody` interface, which allows the application to write directly to the response `OutputStream` without blocking the servlet container thread. According to the Spring documentation, this class is *a controller method return value type for asynchronous*

request processing where the application can write directly to the response OutputStream without holding up the Servlet container thread StreamingResponseBody - Spring Framework 6.2.6 API.

In Spring MVC, `StreamingResponseBody` enables asynchronous writing relative to the request-handling thread, but the underlying I/O remains blocking—specifically the writes to the `OutputStream`. When using Virtual Threads, the I/O operations are more efficient, as they are executed in the context of a lightweight thread. Most of the computation is done in a separate thread from the one that receives each request; we use a thread pool `TaskExecutor` to process requests, allowing the application to scale and handle multiple clients more efficiently as opposed to the default `TaskExecutor` implementation, which tries to create a thread for each request.

However, the Spring MVC implementation does not effectively support PSSR for these templates, as HTML content is not streamed progressively to the client. This is because the response is only sent once the content written to the `OutputStream` exceeds the output buffer size, which defaults to 8KB. As a result, the client receives the response only after the entire HTML content is rendered, defeating the purpose of PSSR in this context. We attempted to reduce the buffer size by configuring it via an `HttpFilter`, but this did not affect the observed behavior.

The Quarkus implementation also uses handlers based on the blocking interface of the `Observable` class. It implements the **StreamingOutput** interface from the JAX-RS specification to enable PSSR, allowing HTML content to be streamed to the client. While `StreamingOutput` also uses blocking I/O, it operates on Vert.x worker threads, which prevents blocking of the event loop. When Virtual Threads are used, the I/O operations are handled efficiently, as they are executed in lightweight threads.

The Quarkus implementation supports PSSR for these templates by configuring the response buffer size in the *application.properties* file. The default buffer size is 8KB, but we reduced it to 512 bytes, which allows the response to be sent to the client progressively as the HTML content is rendered.

5. Results

All the following benchmarks were conducted on a local machine running Ubuntu 22.04 LTS with a 6-core, 12-thread CPU and 32GB of RAM. All tests were conducted on the OpenJDK VM Corretto 21. The JVM was configured with a minimum heap size of 1024MB and a maximum heap size of 16GB.

For both the Apache Bench and JMeter tests, we simulate a 1000-request warmup period for each route with a concurrent user load of 32 users. The warmup period is followed by the actual test period, during which we simulate 256 requests per user, scaling in increments up to 128 concurrent users.

The results are presented in the form of throughput (number of requests per second) for each template engine, with the x-axis representing the number of concurrent users and the y-axis representing the throughput in requests per second.

Since the obtained results for JMeter and Apache Bench show no significant differences, only the JMeter results will be presented.

5.1. Presentations Results

The results in Figure 1 depict the throughput (number of requests per second) for each template engine, with concurrent users ranging from 1 to 128, from left to right. The benchmarks include HtmlFlow using suspendable web templates (HtmlFlow-Susp), Jstachio using Virtual Threads (Jstachio-Virtual), and Thymeleaf using the reactive View Resolver driver (Thymeleaf-Rx). *Sync* and *Virtual* represent the average throughput of the

blocking approaches (i.e., KotlinX, Rocker, Jstachio, Pebble, Freemarker, Trimou, HtmlFlow, and Thymeleaf) when run in the context of a separate coroutine dispatcher or Virtual Threads, respectively.

We show the HtmlFlow-Susp, Jstachio-Virtual, and Thymeleaf-Rx engines separately to observe the performance of the non-blocking engines when using the Reactive, Virtual Thread, and Suspending approaches. The *Sync* and *Virtual* are aggregated due to the similar performance of different engines when using those approaches.

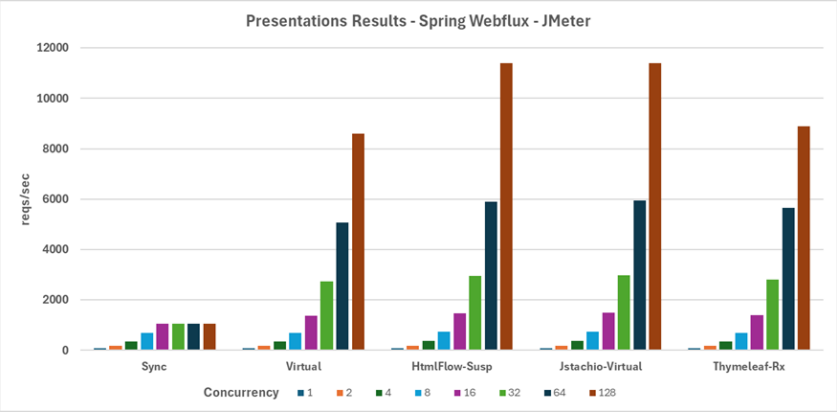

Figure 1. Presentation Benchmark Results in Spring WebFlux with JMeter

The results show that when using the blocking template engines with a separate coroutine dispatcher, the engines are unable to scale effectively beyond 16 concurrent users. In contrast, the non-blocking engines scale effectively up to 128 concurrent users, with HtmlFlow achieving approximately 11,000 requests per second. When using the blocking approaches in the context of Virtual Threads (achieving non-blocking I/O), the engines scale effectively up to 128 concurrent users, with Jstachio matching HtmlFlow's performance at approximately 11,000 requests per second. Thymeleaf, using the reactive View Resolver driver, also scales to 128 users, albeit less effectively, achieving around 8,500 requests per second.

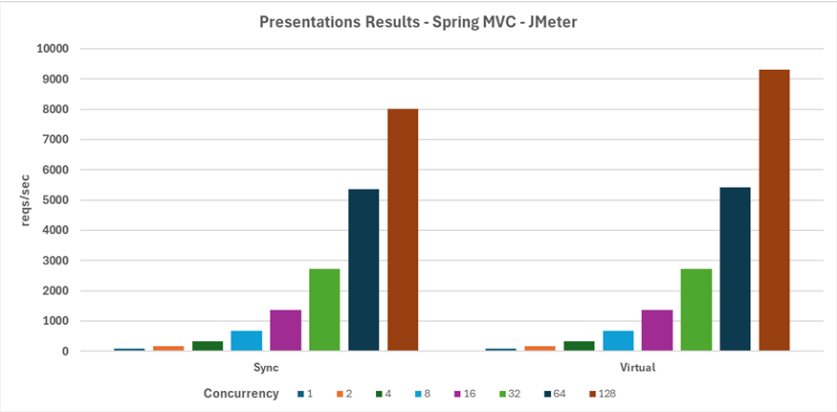

Figure 2. Presentation Benchmark Results in Spring MVC with JMeter

The results for the Spring MVC implementation, shown in Figure 2, compare two approaches: *Sync*, which uses platform threads with `StreamingResponseBody`, and *Virtual*, which uses Virtual Threads. Both approaches scale effectively up to 128 concurrent users, with the Virtual Thread approach achieving a slightly higher throughput of 9,000 requests per second. However, these values are slightly lower than those observed in the Spring WebFlux implementation.

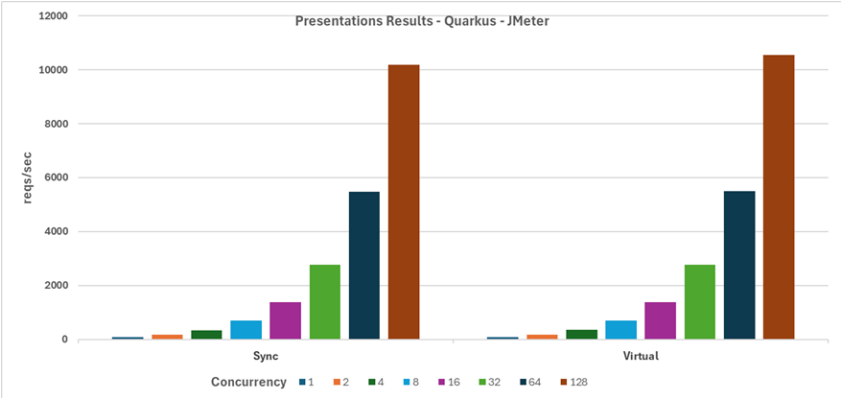

Figure 3. Presentation Benchmark Results in Quarkus with JMeter

The results for the Quarkus implementation, shown in Figure 3 demonstrate that Quarkus handles blocking approaches more effectively than Spring WebFlux, with the blocking engines scaling up to 128 concurrent users and achieving 10,000 requests per second. While the use of Virtual Threads results in a slightly higher throughput, the difference is not significant, as both approaches deliver similar performance.

5.2. Stocks Results

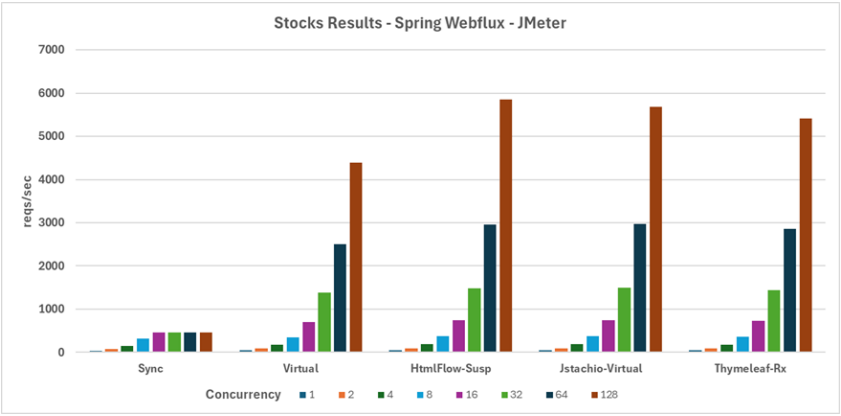

Figure 4. Stocks Benchmark Results in Spring WebFlux with JMeter

The results in Figure 4 use the same template engines and approaches as the previous benchmark, but replace the data model with the more complex Stock class, including 20 instances. Despite the increased complexity and number of instances, the scalability of the engines remains largely unaffected. However, throughput is reduced by approximately 50 percent across all engines, with HtmlFlow achieving 6,000 requests per second. The Thymeleaf implementation using the reactive View Resolver driver reaches 5,000 requests per second, suggesting that the performance drop compared to the Presentation benchmark is not substantial.

The results observed in Figure 5 show that the Spring MVC implementation using the blocking approach with `StreamingResponseBody` achieves a throughput of up to 5500 requests per second, while no significant change is observed when using Virtual Threads. As such, both approaches scale effectively up to 128 concurrent users.

The results depicted in Figure 6 show that the Quarkus implementation scales effectively up to 128 concurrent users, achieving performance comparable to the Spring WebFlux implementation. The blocking engines reach 6,000 requests per second. Again, there is no significant difference between the Virtual Threads and platform threads approaches, with both achieving similar results.

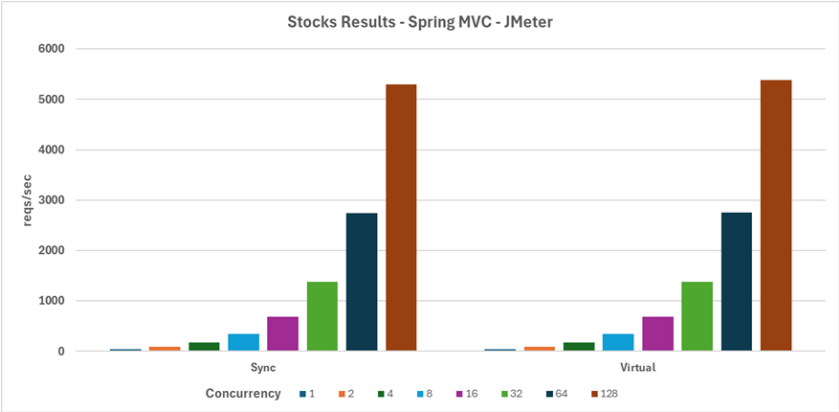

Figure 5. Stocks Benchmark Results in Spring MVC with JMeter

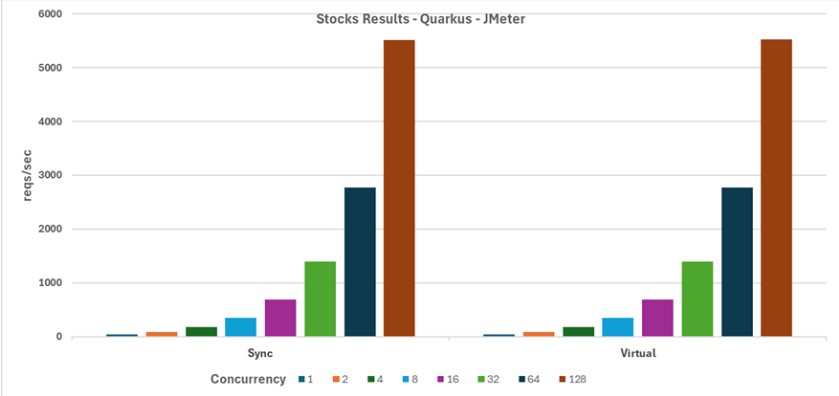

Figure 6. Stocks Benchmark Results in Quarkus with JMeter

The results of the benchmarks show that non-blocking engines, through the use of reactive programming, Kotlin coroutines, or Java virtual threads, are able to scale effectively up to 128 concurrent users. Out of all the tested frameworks, Spring Webflux showed itself the most effective at enabling PSSR, mostly due to its native support for Publish and Subscriber interfaces, which allow for HTML content to be progressively streamed to the client. Quarkus also enabled PSSR effectively, but it required additional configuration of the `OutputBuffer` size to achieve the same results as Spring Webflux. The Spring MVC implementation, on the other hand, did not enable PSSR for the tested templates.

Additionally, the results show that approaches using Virtual Threads are able to scale as effectively as those using reactive programming or Kotlin coroutines, with the advantage of being easier to implement and understand for developers.

6. Conclusion

Through benchmarking across Spring WebFlux, Spring MVC, and Quarkus, we evaluated eight different template engines and found that non-blocking implementations—especially those using virtual threads—consistently deliver performance on par with traditional blocking approaches under high concurrency. These results highlight virtual threads as a promising alternative to complex asynchronous programming models, offering a simpler development experience without sacrificing scalability or responsiveness.

Author Contributions: 'Conceptualization, X.X. and Y.Y.; methodology, X.X.; software, X.X.; validation, X.X., Y.Y. and Z.Z.; formal analysis, X.X.; investigation, X.X.; resources, X.X.; data curation, X.X.; writing—original draft preparation, X.X.; writing—review and editing, X.X.; visualization, X.X.; supervision, X.X.; project administration, X.X.; funding acquisition, Y.Y. All authors have read and agreed to the published version of the manuscript., please turn to the CRediT taxonomy for the term explanation. Authorship must be limited to those who have contributed substantially to the work reported.

Funding: This research received no external funding

Institutional Review Board Statement: Not applicable.

Data Availability Statement: We encourage all authors of articles published in MDPI journals to share their research data. In this section, please provide details regarding where data supporting reported results can be found, including links to publicly archived datasets analyzed or generated during the study. Where no new data were created, or where data is unavailable due to privacy or ethical restrictions, a statement is still required. Suggested Data Availability Statements are available in section "MDPI Research Data Policies" at https://www.mdpi.com/ethics.

Conflicts of Interest: Declare conflicts of interest or state "The authors declare no conflicts of interest." Authors must identify and declare any personal circumstances or interest that may be perceived as inappropriately influencing the representation or interpretation of reported research results. Any role of the funders in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript; or in the decision to publish the results must be declared in this section. If there is no role, please state "The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results".

References

1. Spring WebFlux Comparing Template Engines. 2025. *GitHub*. Available online: https://github.com/xmlet/spring-webflux-comparing-template-engines.
2. James Bucanek. 2009. Model-View-Controller Pattern. In *Learn Objective-C for Java Developers*; Apress: Berkeley, CA, USA; pp. 353-402.
3. Spring Framework. 2025. StreamingResponseBody - Spring Framework 6.2.6 API. Available online: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/StreamingResponseBody.html.