MDPI

Article

Enabling Progressive Server-Side Rendering for Traditional Web Template Engines with Java Virtual Threads

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: Modern web applications increasingly demand rendering techniques that optimize performance, responsiveness, and scalability. Progressive Server-Side Rendering (PSSR) bridges the gap between Server-Side Rendering and Client-Side Rendering by progressively streaming HTML content, improving perceived load times. However, traditional HTML template engines often rely on blocking interfaces that hinder their use in asynchronous, non-blocking contexts required for PSSR. This work explores how Java Virtual Threads, introduced in Java 21, enable non-blocking execution of blocking I/O operations, allowing the reuse of traditional template engines for PSSR without complex asynchronous programming models. We benchmark multiple engines across Spring WebFlux, Spring MVC, and Quarkus using reactive, suspendable, and virtual-thread-based approaches. Results show that virtual threads allow blocking engines to scale comparably to those designed for non-blocking I/O, achieving high throughput and responsiveness under load. This demonstrates that Virtual Threads provide a compelling path to simplify the implementation of PSSR with familiar HTML templates, significantly lowering the barrier to entry while maintaining performance.

Keywords: Web Templates; Server-Side Rendering; Non-Blocking; Java Virtual Threads; Asynchronous

1. Introduction

Modern web applications rely on different rendering strategies to optimize performance, user experience, and *scalability*. The two most dominant approaches are *Server-Side Rendering (SSR)* and *Client-Side Rendering (CSR)*.

SSR generates HTML content on the server before sending it to the client, resulting in faster initial page loads and better Search Engine Optimization (SEO). However, SSR can increase server load and reduce *throughput*[1] since each request requires additional processing before responding.

In contrast, CSR shifts the rendering workload to the browser. The server initially sends a minimal HTML document with JavaScript, which dynamically loads the page content. While CSR reduces the server's burden, it can lead to slower initial load times, as users must wait for JavaScript execution before meaningful content appears.

Progressive Server-Side Rendering (PSSR) combines benefits from both SSR and CSR by streaming HTML content progressively. This technique enhances user-perceived performance by allowing immediate rendering while data loads asynchronously. In this respect, PSSR is similar to CSR in that the server initially sends a minimal HTML document to the

[1] The number of requests the server can handle per second (RPS)

Received:
Revised:
Accepted:
Published:

Citation: . Enabling Progressive Server-Side Rendering for Traditional Template Engines with Java Virtual Threads. *Software* **2025**, *1*, 0. https://doi.org/

Copyright: © 2025 by the authors. Submitted to *Software* for possible open access publication under the terms and conditions of the Creative Commons Attri- bution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

client and subsequently streams additional HTML fragments as data becomes available. However, unlike CSR, PSSR retains all rendering responsibilities on the server side, thereby reducing the load on the client. As a result, the client does not need to execute JavaScript or make additional requests to retrieve page content. To enable non-blocking PSSR, template engines require support for asynchronous data models and non-blocking I/O operations. While some template engines, such as HtmlFlow, and Thymeleaf, have support for these features, traditional template engines that use *external* domain-specific languages (DSLs) often rely on blocking interfaces, such as `Iterable`, to define the data model. This reliance on blocking interfaces leads to the system thread being blocked until the entire HTML content is rendered, which can significantly impact performance and scalability.

Efficient thread utilization is crucial for low-thread servers, necessitating asynchronous, non-blocking request handling. In these architectures, asynchronous I/O operations (e.g., database queries, and API calls) allow servers to manage multiple concurrent requests efficiently by avoiding blocking threads while waiting for I/O operations to complete. In these architectures, where idle time during I/O operations is reduced, non-blocking PSSR enhances a server's capacity to handle high request volumes with limited threads and immediately send partial responses to clients.

With the introduction of *Virtual Threads* in Java 21, it is now possible to execute blocking I/O operations in a non-blocking manner, allowing for the use of these traditional template engines in non-blocking contexts without the need for complex asynchronous programming models. This advancement enables the implementation of PSSR using familiar HTML syntax, simplifying template development and adoption.

This work will focus on exploring existing approaches to non-blocking PSSR, which include the use of *reactive programming* and *coroutines* to implement non-blocking I/O operations. As an alternative to these approaches, we will investigate the viability of *Virtual Threads* in Java 21 for implementing non-blocking PSSR. Section 2 provides an overview of the state-of-the-art in non-blocking PSSR and HTML template engines. Section 3 provides a more detailed description of the issues associated with traditional template engines for non-blocking PSSR, and how this work aims to address them. Section 4 describes the benchmark methodology and implementation. Finally, Sections 5 and 6 present the results and conclusions of this work, respectively.

2. Background and Related Work

In this section, we first examine the different design choices adopted by web servers in their internal architectures, and how these choices impact the behavior of the web template engines. Then, in Subsection 2.2, we present the main properties that characterize each web template technology approach, along with the advantages and drawbacks that result from these characteristics.

2.1. Web Framework Architectures and Approaches to PSSR

In traditional thread-per-request architectures, each incoming request is handled by a dedicated thread. The web server maintains a thread pool from which a thread is assigned to handle each request. As the load increases, the number of active threads can grow rapidly, potentially exhausting the pool. This can lead to performance and scalability issues, as the system may become bogged down by context switching and thread management overhead [1]. Spring MVC is one of the most widely used frameworks that follows this model.

On the other hand, in modern *low-thread* architectures, the server uses a small number of threads to handle a large number of requests. *Low-thread* servers, also known as *event-*

driven [2], offer a significant advantage in efficiently managing a high number of concurrent I/O operations with minimal resource usage.

The *non-blocking* I/O model employed in low-thread servers is well-suited for handling large volumes of data asynchronously [3]. This combination of low-thread servers and asynchronous data models has facilitated the development of highly scalable, responsive, and resilient web applications capable of managing substantial data loads [4]. The prominence of this concept increased with the advent of Node.js in 2009, and subsequently, various technologies adopted this approach in the Java ecosystem, including Netty, Akka HTTP, Vert.X, and Spring WebFlux. The *non-blocking* I/O model in low-thread servers functions optimally only when HTTP handlers avoid blocking. Therefore, HTML templates need to be proficient in dealing with the asynchronous APIs provided by data models. While most legacy web templates struggle with asynchronous models, DSLs for HTML face no such limitations, leveraging all constructions available in the host programming language. However, the unexpected intertwining of asynchronous handlers' completion and HTML builders' execution may potentially lead to malformed HTML with an unexpected layout, as demonstrated in the subsequent subsection 2.2.3.

An alternative approach involves utilizing user-level threads, while maintaining a blocking I/O and a synchronous programming paradigm. However, this approach still requires a user-level I/O subsystem capable of mitigating system-level blocking, which is crucial for the performance of I/O-intensive applications. This technique offers a lightweight solution for efficiently managing a larger number of concurrent sessions by minimizing per-thread overhead. In 2020, Karsten [5] demonstrated how this strategy supports a synchronous programming style, effectively abstracting away the complexities associated with managing continuations in asynchronous programming. A similar idea has been followed in Kotlin with *coroutines* [6], and most recently in the Java standard library with *virtual threads* released in Java 21. However, no work has yet leveraged this mechanism to enable legacy template engines to provide PSSR while preserving non-blocking progress.

2.1.1. Spring WebFlux

2.1.2. Spring MVC

2.1.3. Quarkus

2.2. Web Templates

Web templates have been the most widely adopted approach for constructing dynamic HTML pages. Web templates (such as JSP, Handlebars, or Thymeleaf), also known as *web views* [7,8], are based on HTML documents augmented with template-specific markers (e.g., `<%>`, `{{}}`, or `${}`), which represent *dynamic* information to be replaced at runtime with the results of corresponding computations, producing the final HTML page. The process of parsing and replacing these markers—i.e., *resolution*—is the primary responsibility of the *template engine* [9].

One key characteristic of *web templates* is their ability to receive a *context object*—equivalent to the *model* in the model-view design pattern [9,10]—which provides the data used to fill template placeholders at runtime.

Web templates can be distinguished by several properties, namely:

1. Templating idiom
2. Supported data model APIs
3. Asynchronous support
4. Type safety and HTML safety
5. Progressive rendering

Although some of the aforementioned characteristics apply to both server-side and client-side approaches, we focus solely on web template technologies for server-side rendering, as our work is centered on that approach.

Before detailing each of the aforementioned characteristics, Table 11 presents a breakdown of state-of-the-art template engines, classified according to the identified properties.

2.2.1. Templating idiom

Web templates are based on a *domain-specific language* (DSL) [11], which defines a language tailored to a specific *domain* [12]—in this case, HTML for expressing web documents. The DSL constrains the template's syntax and semantics to match the structure and purpose of HTML.

DSLs can be divided in two types: *external* or *internal*[13]. *External* DSLs are languages created without any affiliation to a concrete programming language. An example of an *external* DSL is the regular expressions search pattern[14], since it defines its own syntax without any dependency of programming languages. On the other hand an *internal* DSL is defined within a host programming language as a library and tends to be limited to the syntax of the host language, such as Java. JQuery[15] is one of the most well-known examples of an internal DSL in Javascript, designed to simplify HTML DOM[16] tree traversal and manipulation.

Traditionally, web template technologies use an *external* DSL to define control flow constructs and data binding primitives. Early web template engines such as JSP, ASP, Velocity, PHP, and others adopted this *external* DSL approach as their templating dialect. For example, a `foreach` loop can be expressed in each technology using its own DSL: `<% for(String item : items) %>` in JSP, `<% For Each item In items %>` in legacy ASP with VBScript, `#foreach($item in $items)` in Velocity, or `<?php foreach ($items as $item):?>` in PHP.

On the other hand, *internal* DSLs for HTML allow templates to be defined directly within the *host* language (such as Java, Kotlin, Groovy, Scala, or other general-purpose programming languages), rather than using text-based template files [17]. In this case, a web template is not limited to templating constructs but may instead leverage any available primitive of the host language or any external API.

For example, in Java, you may iterate over data using a `for` statement, or using the `java.util.streams` API, or any other JVM library such as Guava, Varv, StreamEx, or others—according to your preferences.

Using an internal DSL can have several benefits over using textual templates:

1. *Type safety*: Because the templates are defined with the host programming language, the compiler can check the syntax and types of the templates at compile time, which can help catch errors earlier in the development process.
2. *IDE support*: Many modern IDEs provide code completion, syntax highlighting, and other features, which can make it easier to write and maintain templates.
3. *Flexibility*: Use all the features of the host programming language to generate HTML, can make it easier to write complex templates and reuse code.
4. *Integration*: Because the templates are defined in Java code, for example, you can easily integrate them with other Java code in your application, such as controllers, services, repositories and models.

DSLs for HTML provide an API where methods or functions correspond to the names of available HTML elements. These methods, also known as *builders*, can be combined in a chain of calls to mimic the construction of an HTML document in a fluent manner. Martin Fowler[13] identifies three different patterns for combining functions to create a

DSL: 1) *function sequence*; 2) *nested function*, and 3) *method chaining*, which are illustrated in the snippets of Figure 1.

```
html();
 head();
  title();text("JT");end();
 end();
 body();
  p();
   text("Hi␣JATL");
  end();
 end();
end();
```

(a) Function sequence

```
html(
 head(
  title("ST")
 ),
 body(
  p("Hi␣ScalaTags")
 )
);
```

(b) Nested function

```
html()
 .head()
  .title().text("HF").__()
 .__()
 .body()
  .p()
   .text("Hi␣HtmlFlow")
  .__()
 .__()
.__();
```

(c) Method chaining

Figure 1. Utilizing DSL for HTML libraries with JATL, ScalaTags, and HtmlFlow.

From the three examples depicted in Figure 1, the *nested function* idiom used by **ScalaTags**, as shown in the snippet of Figure 1.b), is the least verbose, requiring fewer statements than the other two cases. This approach of combining function calls is also utilized by **Hiccup** Clojure Library and **j2html** Java library. One reason for verbosity in *function sequence* and *method chaining* approaches is the requirement of a dedicated HTML builder to emit the closing tag, exemplified by `end()` in **JATL** (Figure 1.a) and `__()` in **HtmlFlow** (Figure 1.c). But the *nested function* approach comes with a notable drawback: **they do not support PSSR** because the sequence of nested functions is evaluated backward to the order in which they are written. In other words, arguments are evaluated before the functions are invoked. Taking the example in Figure 1.b), the `title()` function is first evaluated, and its resulting paragraph becomes the argument for the `head()` call, which, in turn, becomes the argument for `html()`, and so on. If HTML is emitted as functions are called, it will print tags in reverse order. The aforementioned DSLs must collect resulting nodes into an internal data structure, which is later traversed to produce the HTML output. Therefore, they cannot progressively emit the output as builders are called.

Two other JVM libraries, **Groovy MarkupBuilder** and **KotlinX.html**, also adopt a *nested function* approach, but they address the backward evaluation issue by implementing *lazy evaluation* of arguments [18], expressed in lambda expressions (also known as function literals). Due to the concise form of expressing lambdas with brackets (i.e., `{}`) in both Groovy and Kotlin, translating the template shown in Figure 1.b) to use Groovy MarkupBuilder or KotlinX.html only requires replacing parentheses with brackets for parent elements. **HtmlFlow** also adopts this approach and provides a Kotlin-idiomatic API as an alternative to its original Java API.

2.2.2. Supported data model APIs

Restricted or unrestricted.

2.2.3. Asynchronous support

Explicar Observable e modelos reactivos

Distinguish between template engines that support and do not support asynchronous data models.

1. Nem se quer Existe API para iterar sobre um Async data model. E.g JStachio, ... apenas suportam interface Iterable. => Compilation Error.

2. Consegue usar qq API e.g Kotlinx.Html and Groovy => usa mal => produzir HTML out of order

3. Sim suporta => well-formed HTML => Thymleaf e o HtmlFlow

One of the reasons for legacy web templates not supporting asynchronous APIs is the absence of a unified standard calling convention for asynchronous calls. While there is a single, straightforward way to use a synchronous API with a direct style, where the result

of a method call corresponds to its returned value, there is no equivalent standard in the asynchronous approach. Instead, we may encounter various asynchronous conventions depending on the programming language and runtime environment. Some of these approaches include *continuation-passing style* (CPS) [19], *promises* [20] async/await idiom [21], reactive streams [22], Kotlin Flow [23], and others.

Many *general-purpose languages* (GPLs) have embraced the `async/await` feature [21] enabling non-blocking routines to mimic the structure of synchronous ones, allowing developers to reason about instruction flow sequentially. The simplicity and broad adoption of this programming model have led to its incorporation into mainstream languages like C#, JavaScript, Python, Perl, Swift, Kotlin, and others, excluding Java. However, implementing `async/await` requires compiler support to translate *suspension points* (i.e., `await` statements) into state machines. Most template engines operate using an external DSL with their own templating dialect (e.g., Thymeleaf, JSP, Jade, Handlebars, and others), which do not inherently leverage asynchronous capabilities from their host GPLs.

2.2.4. Type safety and HTML safety

Another key characteristic of such DSLs is **HTML safety**, which refers to whether they produce only valid HTML conforming to a well-formed document. To ensure HTML safety, a DSL API should only allow combining calls to builders that result in valid HTML. However, most HTML DSLs using the *function sequence* or *nested function* approach cannot ensure HTML safety at compile time.

The *function sequence* approach, as illustrated in the snippet of Figure 1.a), involves combining function calls as a sequence of statements, making it challenging to restrict the order of statements. In the *nested function* approach shown in Figure 1.b), variable-length arguments are often used to allow an undefined number of child elements, which cannot be strongly typed in every programming language.

KotlinX.html, which utilizes the *nested function* idiom with lazy evaluation, mitigates this issue through function types with a receiver. In this approach, the receiver (i.e., `this` within the lambda) is strongly typed and provides a set of methods corresponding to legal child elements. This enables KotlinX.html to enforce HTML safety by restricting the available methods during compile time.

To achieve this KotlinX.html provides HTML builders using *function literals with receiver* [23]. In Kotlin, a block of code enclosed in curly braces `{...}` is known as a *lambda*, and can be used as an argument to a function that expects a *function literal*. When we write, for example, `body { div { hr() } }`, we are invoking the `body` function with a lambda as its argument. This lambda, in turn, calls the `div` function with another lambda as an argument that creates a horizontal row (i.e. `hr`). Each call to an HTML builder (e.g., `body`, `div`, `hr`) creates the child element within the element generated by the outer function call.

HtmlFlow provides two APIs: one in idiomatic Kotlin, similar to the `KotlinX.html` API, and another that employs the *method chaining* idiom, as illustrated in the snippet of Figure 1.c). In this approach, the receiver object is implicitly passed as an argument to each method call, enabling subsequent methods to be invoked on the result of the preceding one. This facilitates the composition of methods, with each call building upon the other. Similar to KotlinX.html, HtmlFlow ensures HTML safety, restricting the available HTML builders and attributes after the dot (`.`) operator.

2.2.5. Progressive rendering

3. Problem Statement

In this section we discuss the problems associated with progressive server-side rendering (PSSR) in modern web applications, focusing on the challenges and limitations of

existing approaches. We then present the objective of our work, which aims to increase the available options for PSSR in web applications, particularly in the context of JVM based frameworks.

*A*s we outlined in the previous section, *external* DSLs for HTML such as Thymeleaf, Jstachio, Handlebars, and others define HTML templates within HTML documents with markers specific to the engine. Template engines that use *internal* DSLs, such as KotlinX or HtmlFlow, define the HTML templates in the programming language itself, leveraging the language's syntax and features to create the templates.

Internal DSLs can more easily support asynchronous data models since they can leverage the language's existing features and libraries, thus enabling the progressive streaming of HTML content to the client, with *external* DSLs supporting simpler blocking data models such as `Iterable` or `List`. While on the previous section we outlined several advantages of using *internal* DSLs over textual templates for HTML, there are also advantages to using *external* DSLs:

1. *Separation of Concerns*: *External* DSLs allow for a clear separation between the HTML structure and the application logic, making it easier to maintain and update templates without modifying the underlying code. This " separation" can be beneficial, as it allows front-end developers to work on the HTML templates without needing to understand the underlying application logic or programming language.
2. *Cross-Language Compatibility*: *External* DSLs can be used across different programming languages and frameworks, making them more versatile and easier to integrate into existing projects. This is particularly useful in environments where multiple languages are used.
3. *Familiarity*: Many developers are already familiar with HTML and its syntax, making it easier to work with *external* DSLs that use HTML-like syntax.

Due to these advantages, many developers prefer to use *external* DSLs for HTML templates, even if it means sacrificing some of the advantages of using *internal* DSLs. However, this preference can lead to challenges when it comes to progressive server-side rendering (PSSR) and non-blocking I/O, as many *external* DSLs do not support asynchronous data models.

?? shows the HTML template for a `Presentation` class using the JStachio template engine, while **??** presents the equivalent template using HtmlFlow.

```
<!DOCTYPE html>
<html lang="en-US">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <title>
        JFall 2013 Presentations - JStachio
    </title>
    <link rel="Stylesheet" href="/webjars/bootstrap/5.3.0/css/bootstrap.min.css">
</head>
<body>
<div class="container">
    <div class="pb-2 mt-4 mb-3 border-bottom">
        <h1>
            JFall 2013 Presentations - JStachio
        </h1>
    </div>
    {{#presentationItems}}
    <div class="card mb-3 shadow-sm rounded">
        <div class="card-header">
            <h5 class="card-title">
                {{title}} - {{speakerName}}
            </h5>
        </div>
        <div class="card-body">
```

```
            {{summary}}
        </div>
    </div>
    {{/presentationItems}}
</div>
</body>
</html>
```

Listing 1: Presentation HTML template using *JStachio* label

```
val htmlFlowTemplate: HtmlViewAsync<Observable<Presentation>> =
    HtmlFlow.viewAsync<Observable<Presentation>?> { view ->
        view.html {
            attrLang("en-us")
            head {
                meta { attrCharset("UTF-8") }
                meta { attrName("viewport").attrContent("width=device-width, initial-
                    scale=1.0") }
                meta { addAttr("http-equiv", "X-UA-Compatible").attrContent("IE=Edge") }
                title { text("JFall 2013 Presentations - HtmlFlow") }
                link { attrRel(EnumRelType.STYLESHEET).attrHref("/webjars/bootstrap
                    /5.3.0/css/bootstrap.min.css") }
            }
            body {
                div {
                    attrClass("container")
                    div {
                        attrClass("pb-2 mt-4 mb-3 border-bottom")
                        h1 { text("JFall 2013 Presentations - HtmlFlow") }
                    }
                    await { div, model: Observable<Presentation>, onCompletion ->
                        model
                            .doOnNext {
                                presentationFragmentAsync
                                    .renderAsync(it)
                                    .thenApply { frag -> div.raw(frag) }
                            }
                            .doOnComplete { onCompletion.finish() }
                            .subscribe()
                    }
                }
            }
        }
    }.threadSafe()
```

Listing 2: Presentation HTML template using *HtmlFlow* label

HtmlFlow is an example of an *internal* DSL that uses calls to HTML builders as a way to emit HTML. It allows for the use of the `Observable` class as a data model, which in turn allows for the definition of a `Consumer<Presentation>` through the `doOnNext` method. Every time a new value is emitted to the `Observable`, the `doOnNext` method is called, emitting the HTML fragment for the `Presentation` object to the client.

The Jstachio template engine, on the other hand, uses an *external* DSL, where the HTML template is defined in a separate file. The template engine uses the `presentationItems` variable to define the data model, which is a list of `Presentation` objects. This allows the template engine to iterate over the list of `Presentation` objects and render the HTML for each one.

As demonstrated, the HtmlFlow template enables PSSR by directly supporting the use of the `Observable` class as a data model. However, not all programmers are familiar with asynchronous programming models, which can make implementing this solution challenging. In contrast, Jstachio defines templates using standard HTML syntax with specific tags for the template engine, resulting in a simpler and more accessible template definition. However, because Jstachio templates rely on an `Iterable` interface, they block the calling thread until the entire HTML content is rendered and therefore do not support non-blocking PSSR.

In previous work [24], Carvalho et al. developed a benchmark using Spring Webflux to evaluate the performance of HtmlFlow with suspendable web templates. In this benchmark, HtmlFlow's performance was compared to that of Thymeleaf using a reactive approach, as well as other template engines such as the one shown in **??**. The results demonstrated that HtmlFlow scaled efficiently up to 128 concurrent users, achieving a throughput of up to 4,000 requests per second. In contrast, the blocking template engines ceased to scale beyond 4 concurrent users, reaching a maximum throughput of approximately 400 requests per second. These findings indicate that engines which block the calling thread scale poorly and are not suitable for PSSR.

With the advent of Java 21 and the introduction of *Virtual Threads*, it is now possible to execute code that would otherwise block the calling thread in a scalable, non-blocking manner. This advancement enables the use of these *external* DSLs in non-blocking contexts. The objective of this work is to evaluate the performance of these template engines when leveraging Virtual Threads, and to determine whether they can be made viable for PSSR. We will also compare the performance of this approach with existing solutions based on reactive programming and Kotlin coroutines.

If the performance of these engines proves comparable to that of current approaches, it would enable PSSR to be implemented using more familiar HTML syntax, simplifying template development and adoption. Furthermore, template engine developers seeking to support PSSR would no longer need to implement complex asynchronous programming models to achieve non-blocking I/O; instead, they could rely on the traditional approaches used for Server Side Rendering (SSR).

4. Benchmark Implementation

The benchmark is designed with a modular architecture, separating the *view* and *model* layers from the *controller* layer [25], 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. It includes two different data models, defined as `Presentation` and `Stock`, shown in Listing 3 and Listing 4, respectively. The `Presentation` class represents a presentation with a title, speaker name, and summary, while the `Stock` class represents a stock with a name, URL, symbol, price, change, and ratio.

```
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 3: `Stock` class

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

Listing 4: `Presentation` class

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 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 WebFlux core 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. The implementation includes three main approaches to PSSR:

- **Reactive**: The template engine is used in a reactive context, where the HTML content rendered using the reactive programming model. An example of this approach is the Thymeleaf template engine when using the `ReactiveDataDriverContextVariable` in conjunction with a non-blocking Spring `ViewResolver`.
- **Virtual**: The template engine is used in a non-blocking context, where the HTML content is rendered within the context of Virtual Threads. This method is used for the template engines that do not traditionally support non-blocking I/O, be it either because they use external DSLs and in consequence only support the blocking `Iterable` interface, or because they do not support the asynchronous rendering of HTML content.
- **Suspendable**: The template engine is used in a suspendable context, where the HTML content is rendered within the context of a suspending function An example of this approach is the HtmlFlow template engine, which supports suspendable templates with the use of the `Flow` class from the Kotlin standard library.

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* [26].

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 when compared to platform threads, 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.

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.

Both the Quarkus and Spring MVC implementations were configured with an output buffer size of 8KB, despite the Quarkus implementation not enabling PSSR at this size. This configuration was chosen to ensure consistency across Spring MVC and Quarkus, as the Spring MVC implementation does not support PSSR for the tested templates.

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 2 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 with the `Iterable` interface (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 Suspending, Virtual Thread, and Reactive approaches. The *Sync* and *Virtual* are aggregated due to the similar performance of different engines when using those approaches.

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.

The results for the Spring MVC implementation, shown in Figure 3, compare two approaches: *Sync*, which uses platform threads with `StreamingResponseBody`, and *Virtual*,

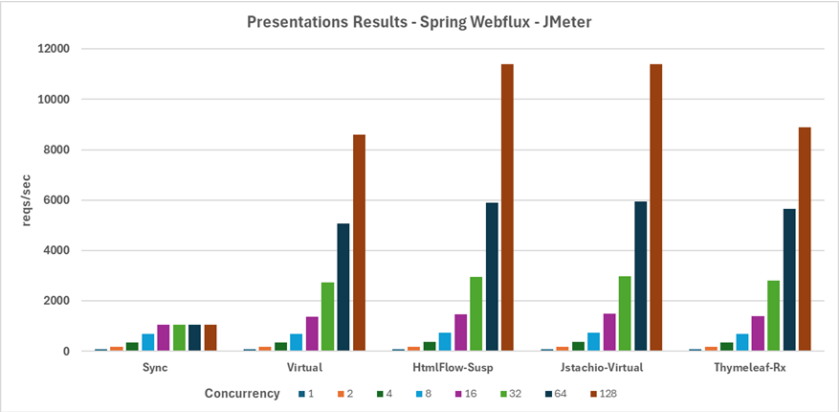

Figure 2. Presentation Benchmark Results in Spring WebFlux with JMeter

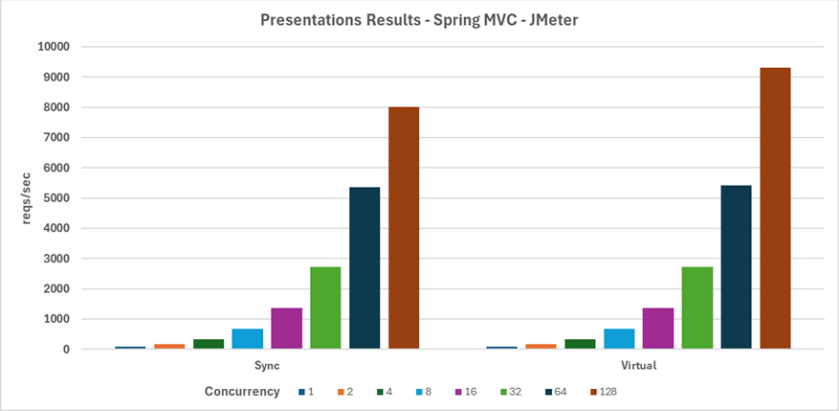

Figure 3. Presentation Benchmark Results in Spring MVC with JMeter

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.

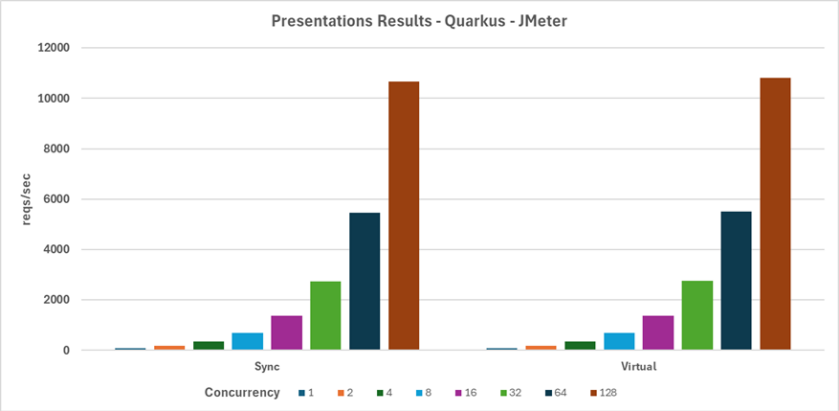

Figure 4. Presentation Benchmark Results in Quarkus with JMeter

The results for the Quarkus implementation, shown in Figure 4 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.

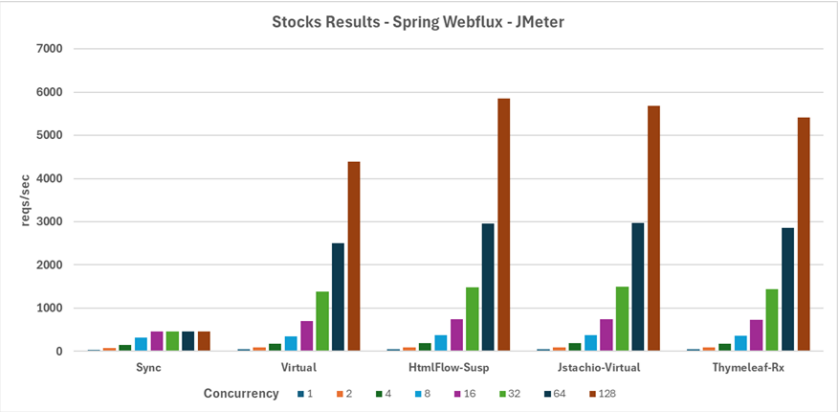

Figure 5. Stocks Benchmark Results in Spring WebFlux with JMeter

5.2. Stocks Results

The results in Figure 5 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. Since the `Stock` class contains more data properties than the `Presentation` class, the data binding to the template engine adds overhead, which explains the reduced performance. 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.

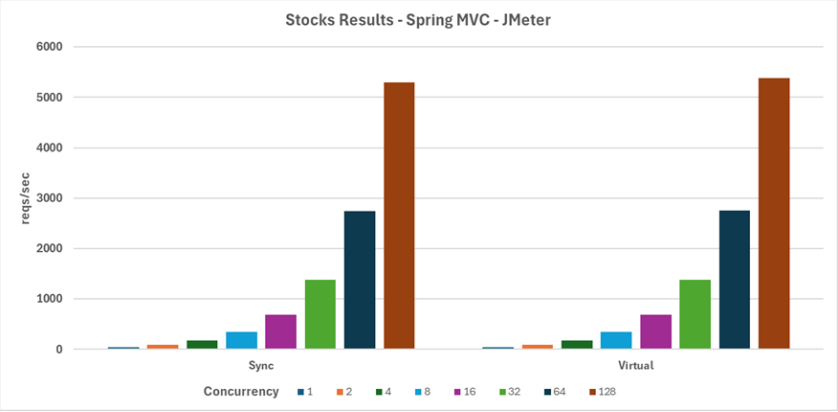

Figure 6. Stocks Benchmark Results in Spring MVC with JMeter

The results observed in Figure 6 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 7 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.

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

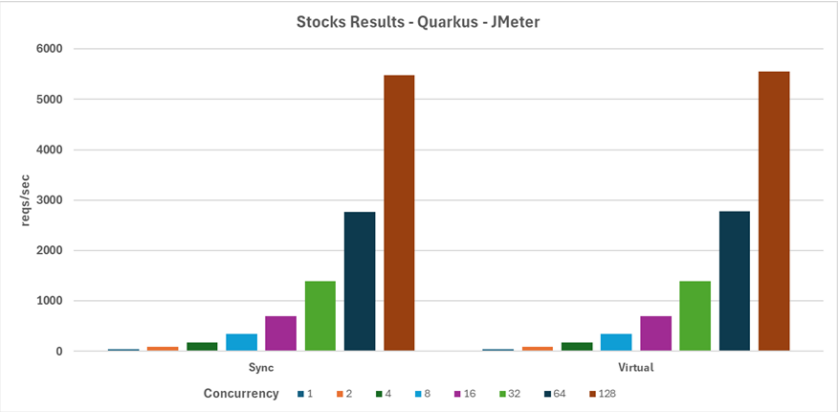

Figure 7. Stocks Benchmark Results in Quarkus with JMeter

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, allowing *external* DSLs to be used in a non-blocking context, effectively enabling PSSR.

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. Kant, K.; Mohapatra, P. Scalable Internet servers: Issues and challenges. *ACM SIGMETRICS Performance Evaluation Review* **2000**, *28*, 5–8.
2. Elmeleegy, K.; Chanda, A.; Cox, A.L.; Zwaenepoel, W. Lazy Asynchronous I/O for Event-Driven Servers. In Proceedings of the Proceedings of the Annual Conference on USENIX Annual Technical Conference, USA, 2004; ATEC '04, p. 21.
3. Meijer, E. Your Mouse is a Database. *Queue* **2012**, *10*, 20:20–20:33. https://doi.org/10.1145/2168796.2169076.
4. Jin, X.; Wah, B.W.; Cheng, X.; Wang, Y. Significance and Challenges of Big Data Research. *Big Data Res.* **2015**, *2*, 59–64. https://doi.org/10.1016/j.bdr.2015.01.006.
5. Karsten, M.; Barghi, S. User-Level Threading: Have Your Cake and Eat It Too. *Proc. ACM Meas. Anal. Comput. Syst.* **2020**, *4*. https://doi.org/10.1145/3379483.
6. Elizarov, R.; Belyaev, M.; Akhin, M.; Usmanov, I. Kotlin coroutines: design and implementation. In Proceedings of the Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2021, pp. 68–84.
7. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
8. Alur, D.; Malks, D.; Crupi, J. *Core J2EE Patterns: Best Practices and Design Strategies*; Prentice Hall PTR: Upper Saddle River, NJ, USA, 2001.
9. Parr, T.J. Enforcing Strict Model-view Separation in Template Engines. In Proceedings of the Proceedings of the 13th International Conference on World Wide Web, New York, NY, USA, 2004; WWW '04, pp. 224–233. https://doi.org/10.1145/988672.988703.
10. E. Krasner, G.; Pope, S. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk80 System. *Journal of Object-oriented Programming - JOOP* **1988**, *1*, 26–49.
11. Landin, P.J. The next 700 programming languages. *Communications of the ACM* **1966**, *9*, 157–166.
12. Evans, E.; Fowler, M. *Domain-driven Design: Tackling Complexity in the Heart of Software*; Addison-Wesley, 2004.
13. Fowler, M. *Domain Specific Languages*, 1st ed.; Addison-Wesley Professional, 2010.
14. Thompson, K. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* **1968**, *11*, 419–422. https://doi.org/10.1145/363347.363387.
15. Resig, J. *Pro JavaScript Techniques*; Apress, 2007.
16. Hors, A.L.; Hégaret, P.L.; Wood, L.; Nicol, G.; Robie, J.; Champion, M.; Arbortext.; Byrne, S. Document Object Model (DOM) Level 3 Core Specification. Technical report, https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/, 2004.

17. Carvalho, F.M.; Duarte, L. HoT: Unleash Web Views with Higher-order Templates. In Proceedings of the Proceedings of the 15th International Conference on Web Information Systems and Technologies, 2019, WEBIST '19, pp. 118–129. https://doi.org/10.5220/0008167701180129.
18. Landin, P.J. Correspondence Between ALGOL 60 and Church's Lambda-notation: Part I. *Commun. ACM* **1965**, *8*, 89–101. https://doi.org/10.1145/363744.363749.
19. Sussman, G.; Steele, G. *Scheme: an interpreter for extended lambda calculus*; AI Memo No, MIT, Artificial Intelligence Laboratory, 1975.
20. Friedman.; Wise. Aspects of Applicative Programming for Parallel Processing. *IEEE Transactions on Computers* **1978**, *C-27*, 289–296. https://doi.org/10.1109/TC.1978.1675100.
21. Syme, D.; Petricek, T.; Lomov, D. The F# Asynchronous Programming Model. In Proceedings of the Practical Aspects of Declarative Languages; Rocha, R.; Launchbury, J., Eds., Berlin, Heidelberg, 2011; pp. 175–189.
22. Netflix.; Pivotal.; Red Hat.; Oracle.; Twitter.; Lightbend. Reactive Streams Specification. Technical report, https://www.reactive-streams.org/, 2015.
23. Breslav, A. Kotlin Language Documentation. Technical report, https://kotlinlang.org/docs/kotlin-docs.pdf, 2016.
24.
25. Model-View-Controller Pattern. In *Learn Objective-C for Java Developers*; Apress: Berkeley, CA, 2009; pp. 353–402. https://doi.org/10.1007/978-1-4302-2370-2_20.
26. Streaming Response Body, 2025.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.