Article

Enabling Progressive Server-Side Rendering for Legacy 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

Received:
Revised:
Accepted:
Published:

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

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 a faster *First Contentful Paint (FCP)* [1] 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 a slower *FCP*, 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 progressive rendering as data becomes available. In this respect, PSSR is similar to CSR in that the server initially sends a minimal HTML document to the client and subsequently streams additional HTML fragments. However, unlike CSR, PSSR retains all rendering responsibilities on the server side, thereby reducing the load on the client.

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

As a result, the client does not need to execute JavaScript or make additional requests to retrieve page content.

Low-thread servers, also known as *event-driven* [2], have gained prominence in contemporary web applications due to their ability to efficiently manage a large number of concurrent I/O operations with minimal resources, thus promoting better scalability. By leveraging asynchronous I/O operations—such as database queries and API calls, servers can avoid blocking threads while waiting for data, thereby maximizing throughput and responsiveness. To support this non-blocking architecture, PSSR implementations require template engines that are compatible with asynchronous data models [3]. Some modern template engines, such as HtmlFlow [4] and Thymeleaf [5], have been designed with these capabilities in mind. However, many traditional template engines—particularly those using external domain-specific languages (DSLs) [6]—still depend on blocking interfaces like `Iterable` for data processing. This blocking behavior forces server threads to remain idle until the entire HTML output is ready, undermining the performance benefits of non-blocking I/O and limiting scalability in high-concurrency environments.

With the introduction of *Virtual Threads* [2] in Java 21, it is now possible to execute blocking I/O operations in a scalable, lightweight manner. This capability allows traditional template engines—often reliant on blocking interfaces—to operate efficiently in high-concurrency, non-blocking environments without requiring complex asynchronous programming models. As a result, PSSR can now be implemented using familiar HTML templates, simplifying development and improving maintainability.

This work explores the current landscape of non-blocking PSSR, focusing on two primary paradigms: *reactive programming* and *coroutines*, both of which have been used to achieve asynchronous I/O in the Java ecosystem. As an alternative, we investigate whether Java's Virtual Threads can offer comparable performance while preserving the simplicity of synchronous code. Section 2 reviews the state-of-the-art in PSSR and template engine design. Section 3 outlines the limitations of conventional engines in asynchronous settings and presents our proposed approach. Section 4 details the benchmark methodology, followed by the results and analysis in Section 5. Conclusions are presented in Section 6.

2. Background and Related Work

In this section, we first present the main properties that characterize each web template technology approach, along with the advantages and drawbacks that result from these characteristics. Then, in Subsection 2.2, we 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.

2.1. Web Templates

Web templates have been the most widely adopted approach for constructing dynamic HTML pages. Web templates or *web views* [7,8] (e.g. JSP, Handlebars, or Thymeleaf), 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:

[2] https://openjdk.org/jeps/444

1. *Domain-specific language* 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 getting deep on each of the aforementioned characteristics, Table 1 presents a breakdown of mainstream template engines, classified according to the identified properties.

Library	DSL idiom	Data Model APIs	Asynchronous Support	Type Safety	HTML Safety	Progressive Rendering
Freemarker	External DSL	`Iterable`	✗	✗	✗	✓
JSP	External DSL	`Iterable`	✗	✗	✗	✗
JStachio	External DSL	`Iterable`	✗	✓ (4)	✗	✓
Pebble	External DSL	`Iterable`	✗	✗	✗	✓
Qute	External DSL	`Iterable`	✗	∗ (5)	✗	✓
Rocker	External DSL	`Iterable`	✗	✗	✗	✓
Thymeleaf	External DSL	`Iterable` `Stream` `Publisher` (1)	`Publisher` (1)	✗	✗	✓
Trimou	External DSL	`Iterable`	✗	✗	✗	✓
Velocity	External DSL	`Iterable` `Sequence` (2)	✗	✗	✗	✓
Clojure Hiccup	Nested Eager	All	✗	✓	✗	✗
Groovy MarkupBuilder	Nested Lazy	All	✗	✓	✗	✓
HtmlFlow	Chaining	All	✓	✓	✓	✓
j2html	Nested Eager	All	✗	✓	✗	✗
KotlinX	Nested Lazy	All	✗	✓	✓ (3)	✓
ScalaTags	Nested Eager	All	✗	✓	✗	✗

Table 1. Comparison of web template technologies in the Java ecosystem.
(1) `Publisher` of Reactive Streams [11]. Limited to a single model per web template.
(2) `kotlin.sequences.Sequence`
(3) Non-safety for HTML attributes.
(4) Enforced via annotation processor that checks templates against typed interfaces at compile time.
(5) Compile-time expression validation available via `@CheckedTemplate` and build-time metadata in Quarkus.

The first half of Table 1 lists template engines that use their own templating dialects, referred to as *external DSLs*. The second half lists Java libraries that provide an *internal DSL*, typically using a *nested* or *chaining* style to build HTML. Note that, by leveraging the host language (e.g., Clojure, Groovy, Java, Kotlin, or Scala) as the templating idiom, the latter impose no restrictions on the data model and fully support all styles of data access APIs. They also benefit from static type checking, which helps ensure *type safety*.

2.1.1. Domain-specific language idiom

Web templates are based on a *domain-specific language* (DSL) [12], which defines a language tailored to a specific *domain* [13]—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*[14]. *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[15], 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[16] is one of the most well-known examples of an internal DSL in Javascript, designed to simplify HTML DOM[17] 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 $items)` in Velocity
- `<?php foreach ($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 [18]. 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.

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[14] 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 [19], 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.

2.1.2. Supported data model APIs

Template engines with dedicated templating dialects typically rely on specific control flow constructs—such as a `foreach`-like statement—to iterate over elements in a data source. In the Java ecosystem, `Iterable` is the common supertype implemented by most data structures and serves as the standard API for iteration. However, other standard interfaces, such as `java.util.Stream`, also represent sequences of elements but are not compatible with `Iterable`. This fragmentation becomes more evident when considering other JVM languages, such as Kotlin, which introduces additional abstractions like the `Sequence` interface—also incompatible with `Iterable`. As a result, template engines based on *external* DSLs must explicitly support each of these interfaces to accommodate the various iteration protocols across languages and libraries.

For example, Thymeleaf supports both Java `Iterable` and `Stream` iteration protocols but is incompatible with Kotlin's `Sequence` interface. Velocity, on the other hand, uses internal reflection-based template processing that allows it to iterate over any type defining an `iterator()` method returning an `Iterator`. This enables support for both Java `Iterable` and Kotlin `Sequence`, but it cannot handle Java `Stream` since `Stream` does not expose a compatible `iterator()` method. The remaining analyzed template engines that use external DSLs support only Java `Iterable` as their sole iteration protocol.

On the other hand, template engines that employ an *internal* DSL do not face such limitations, as they can leverage any available API within the host programming language. This flexibility extends beyond the standard library to include third-party APIs as well. For example, in Java, developers can iterate over data using a traditional `for` loop, the `java.util.stream` API, or external libraries such as Guava, Vavr, StreamEx, and others—depending on their specific needs or preferences.

2.1.3. Asynchronous support

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) [20], *promises* [21], async/await idiom [22], or *suspend functions* [23].

Many *general-purpose languages* (GPLs) have embraced the `async/await` feature [22] 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 programming languages.

An alternative approach to asynchronous programming involves handling data as a sequence of asynchronous events, commonly known as a *reactive stream*. Instead of materializing all items in memory, reactive streams emit values over time, allowing clients to subscribe and consume data incrementally as it becomes available. This idea was first proposed by Meijer [24] in the .NET framework, leading to the development of *Reactive Extensions* (Rx). Subsequently, several alternative implementations emerged in the Java ecosystem, such as Rx Java [25], Project Reactor [26], or the Akka Streams [27] library. Later, due to the proliferation of alternative implementations, the need arose to create a common interface to ensure interoperability between different reactive stream libraries. This led to the formation of a working group that included engineers from notable companies such as Lightbend, Pivotal, and Netflix. The group began defining the *Reactive Streams* specification [11], which established a standard for asynchronous stream processing using non-blocking, backpressure-aware data pipelines. The more recent Mutiny [28], while conforming to the *Reactive Streams* standard, proposes a different approach that avoids complex optimizations, resulting in more straightforward and easier-to-maintain code.

Thymeleaf was one of the first web template engines—and remains the only one among external DSLs for HTML—to support reactive streams based on the `Publisher` model. However, it is limited to using a single `Publisher` within a given template.

2.1.4. 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* [29]. 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.

2.1.5. Progressive rendering

Pursuing the optimal user experience for web users has been a consistent goal since the inception of the World Wide Web. Consequently, various approaches have been explored to deliver the initial meaningful content to the end-user as promptly as possible, while the remaining content *progressively* (or incrementally) loads as the server streams the HTML content.

Progressive rendering and ***progressive loading*** encompass different concepts. The former pertains to the *dynamic* content of a *dynamic web page*, encompassing elements with logic and placeholders that are fulfilled by data from an object *model* constructed at runtime. On the other hand, the latter is associated with *render-blocking resources* such as scripts, stylesheets, and HTML imports, which may hinder the browser from rendering page content to the screen [30]. The notion of *progressive rendering* has typically been aligned with *client-side rendering* (CSR), where a single HTML page with static content is delivered upfront, while the dynamic content is fetched as the data becomes available to complete the web page. However, despite being overlooked, HTTP and browsers were designed from their inception to also support this feature in the context of SSR approaches, which offers the advantage of not being dependent on *rendering-blocking resources*, such as the required JavaScript for CSR. An example of this limitation in SSR web templates was highlighted by Jeff Atwood in 2005, who criticized Microsoft ASP.NET for loading the entire web page into memory before sending any data to the browser [31]. Despite historical critiques and HTML's inherent capabilities, most Web application frameworks, such as ASP.Net, Express.js, Spring, and others, persistently lack support for progressive rendering, leading to the appearance of alternative techniques leveraging client-side JavaScript. Since 2007, various patents have addressed the PSSR issue, with Microsoft's patent enabling the infinite scrolling technique by displaying a single page of results [32], and Yahoo's patent focusing on differentiating elements based on their position relative to the visible portion [33].

Former techniques all describe methods of progressively adding content to a web page depending on client-side JavaScript. Java Thymeleaf, the default SSR template engine for Spring web servers, added support for PSSR in 2018, through the use of a specific non-blocking Spring `ViewResolver` driver and without requiring client-side JavaScript. Carvalho [3] introduced an SSR solution that manages multiple data models and asynchronous APIs while ensuring well-formed HTML, PSSR, and non-blocking template resolution. That solution builds upon HtmlFlow, a Java DSL for HTML, with an internal processing mechanism that streams HTML in chunks as data items become available from data sources, without requiring client-side JavaScript. Like Thymeleaf, their proposal avoids the use of client-side JavaScript; however, unlike Thymeleaf, it supports a wide range of asynchronous APIs and multiple data sources, not limited to the `Publisher`

API. However, the main counter-argument is the non-trivial management of the *resume* callback in continuations [34,35], which is used to linearize the execution flow between asynchronous calls.

2.2. 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 [36]. In the Java ecosystem, Spring MVC is one of the most widely used frameworks that follow this model, according to several reports such as the Stack Overflow 2024 Developer Survey [3].

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 [37]. 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 [38]. 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 [39].

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 [40] demonstrated how this strategy supports a synchronous programming style, effectively abstracting away the complexities associated with managing continuations in asynchronous programming.

Among mainstream technologies, the Kotlin programming language introduces a new abstraction for managing coroutines and provides structured concurrency, which ensures that coroutines are *scoped*, *cancellable*, and *coordinated* within a well-defined lifecycle [23]. Although this model embraces the `async/await` feature [22], enabling non-blocking routines to mimic the structure of synchronous ones, it still lacks a user-level I/O subsystem and relies on an explicit I/O dispatcher tied to a dedicated thread pool that frees worker threads for handling processing tasks. Moreover, web templates using their own templating dialects, such as JSP, Thymeleaf, or Handlebars, are unable to take advantage of such

[3] https://survey.stackoverflow.co/2024/technology#1-web-frameworks-and-technologies

constructs, as they typically support only a limited subset of the host library's API—most commonly just the `Iterable` interface.

Only Java virtual threads, introduced in JEP 444[4], adopt a similar approach to Karsten's proposal, using user-mode threads to preserve a synchronous programming model while still allowing blocking I/O operations without tying up platform threads. When a virtual thread performs a blocking I/O operation, the JVM intercepts the call and transparently parks the virtual thread, freeing the underlying platform thread to perform other tasks. Once the I/O operation completes, the virtual thread is rescheduled on a platform thread and resumes execution. This mechanism is enabled by the JVM's integration with non-blocking system calls at the OS level, allowing developers to write traditional synchronous code while benefiting from the scalability of non-blocking I/O.

2.2.1. Spring MVC

Spring MVC is a synchronous web framework built on the thread-per-request model. When an HTTP request is received, it is handled by a dedicated thread from the server's thread pool (e.g., Tomcat, Jetty, or Undertow). This thread is responsible for executing the full request lifecycle synchronously, including invoking controllers, executing business logic, and rendering the response view. Because the model is blocking, the thread remains occupied throughout the request—even during I/O operations such as database queries or external API calls—which can lead to performance bottlenecks under high concurrency.

Controllers in Spring MVC typically return Java objects or models that are processed by view resolvers in conjunction with template engines like JSP, Thymeleaf, or FreeMarker. These templates generate the full HTML response in one pass, which is then sent to the client only after all necessary data has been gathered and rendered. By default, this architecture limits the ability to perform PSSR, as traditional rendering waits for all data before sending any output. This results in higher latency for clients and less responsive page loads, particularly for data-intensive views. However, Spring MVC provides a mechanism for response streaming via the `StreamingResponseBody`[5] interface, introduced in Spring 4.2. When a controller method returns a `StreamingResponseBody`, Spring writes directly to the `HttpServletResponse` output stream, allowing the server to send parts of the response incrementally as they become available. This is particularly useful for:

- Sending large responses without buffering the entire output in memory.
- Writing dynamic HTML in fragments as data is fetched or computed.
- Reducing Time to First Byte (TTFB) and improving perceived performance.

```
@GetMapping("/stream")
fun handleStream(): StreamingResponseBody? {
    return StreamingResponseBody { outputStream ->
        for (i in 0..9) {
            val htmlFragment = "<p>Chunk " + i + "</p>\n"
            outputStream.write(htmlFragment.toByteArray())
            outputStream.flush() // ensure partial response is sent
            Thread.sleep(500) // simulate delay
        }
    }
}
```

Listing 1: StreamingResponseBody handler in Spring MVC

In the example shown in Listing 1, HTML content is written and flushed in discrete chunks, allowing the client to progressively render the response as it arrives, while the server continues processing additional data. However, this technique has two limitations.

[4] https://openjdk.org/jeps/444
[5] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/StreamingResponseBody.html

First, it does not eliminate the blocking nature inherent in Spring MVC, as the handling thread remains active during the streaming process. Second, it is constrained by the servlet response buffer. By default, most servlet containers (e.g., Tomcat) use a response buffer size of approximately 8KB. Data written to the response output stream is held in this buffer until it reaches capacity. Consequently, if the total size of the initial HTML content is less than the buffer threshold, the server will not transmit any data to the client until the buffer is filled. This buffering behavior introduces a delay in sending the first chunk of content, thereby negating the benefits of progressive rendering. In scenarios where the rendered HTML template does not produce sufficient data to exceed the buffer threshold early, PSSR becomes ineffective, and the client experiences a delayed initial response.

2.2.2. Spring WebFlux

Spring WebFlux is a reactive web framework built on a non-blocking, event-driven architecture that enables efficient handling of a large number of concurrent connections with minimal resource usage. Unlike traditional servlet-based stacks that rely on a thread-per-request model, WebFlux operates on reactive runtimes like Netty, where a small, fixed-size event loop manages all I/O events. Incoming HTTP requests are routed to handler functions or annotated controllers, which return reactive types such as `Mono<T>` and `Flux<T>`—`Publisher` implementations provided by the Project Reactor library [26]. These implementations adhere to the Reactive Streams specification [11], enabling interoperability with other compliant reactive stream libraries. A `Mono` represents a single asynchronous value (or none) [21], while a `Flux` represents a stream of zero or more values [24]. These abstractions allow the entire request lifecycle—from controller invocation to response rendering—to be orchestrated as a chain of non-blocking, asynchronous operations. I/O-bound tasks such as database queries or external API calls do not occupy threads during execution, enabling the framework to scale gracefully under high load and maintain responsiveness even in resource-constrained environments.

PSSR is natively supported in WebFlux via `Flux`-based controllers. When a controller returns a `Flux<String>` or another stream of HTML fragments, WebFlux can begin streaming content to the client as soon as the first elements are emitted.

```java
@GetMapping(value = "/pssr", produces = MediaType.TEXT_HTML_VALUE)
public Flux<String> renderChunks() {
    return Flux.range(1, 10)
               .delayElements(Duration.ofMillis(500))
               .map(i -> "<p>Chunk " + i + "</p>\n");
}
```

Listing 2: Progressive Server-Side Rendering in Spring WebFlux

Listing 2 shows a method that emits one HTML chunk every 500 milliseconds. As each chunk is generated, it is immediately written to the HTTP response, allowing the client to begin rendering content even as more data is still being produced. Unlike `StreamingResponseBody` in Spring MVC, which is constrained by servlet buffer thresholds and blocking semantics, WebFlux ensures that each emitted item is pushed to the client as soon as the network is ready, without occupying a dedicated thread. Moreover, WebFlux integrates seamlessly with reactive data sources such as R2DBC (Reactive Relational Database Connectivity) and reactive NoSQL drivers, making it possible to build end-to-end non-blocking applications. HTML builders and DSLs used in conjunction with WebFlux must be designed to accommodate reactive streams, ensuring that view generation does not violate the non-blocking contract by performing blocking I/O or awaiting asynchronous operations imperatively.

2.2.3. Quarkus

Quarkus is a modern, cloud-native Java framework that supports both imperative and reactive programming models. It is built on top of Vert.x, which provides a non-blocking, event-driven runtime similar to Netty. Incoming HTTP requests are processed asynchronously on a small, fixed-size event loop thread pool, allowing the framework to handle many concurrent connections efficiently without blocking threads. Quarkus provides support for reactive APIs like Mutiny [28], a modern alternative to Spring Reactor [26]. Like Reactor, Mutiny complies with the Reactive Streams specification [11]. For traditional imperative-style request handling, Quarkus provides integration with JAX-RS, where resource methods can return synchronous types or asynchronous constructs such as `CompletionStage`. To enable PSSR in imperative endpoints, Quarkus leverages the JAX-RS `StreamingOutput` interface. By returning a `StreamingOutput` instance, developers can write parts of the HTTP response body incrementally as data becomes available. This approach allows the server to flush HTML fragments progressively to the client, improving the perceived responsiveness especially for long-running or large responses. However, like other streaming mechanisms based on servlet buffers, the initial data flush may be delayed until internal buffers (commonly around 8KB) are filled. However, Quarkus allows configuring the buffer size to a smaller value, which can help mitigate this issue.

```java
@GET
@Path("/stream")
@Produces(MediaType.TEXT_HTML)
public StreamingOutput streamHtml() {
    return output -> {
        PrintWriter writer = new PrintWriter(output);
        for (int i = 1; i <= 5; i++) {
            writer.println("<p>Chunk " + i + "</p>");
            writer.flush(); // Flush each chunk incrementally
            Thread.sleep(500); // Simulate processing delay
        }
        writer.flush();
    };
}
```

Listing 3: Progressive Server-Side Rendering in Quarkus using StreamingOutput

Listing 3 shows how `StreamingOutput` can be used to send partial HTML content in chunks, allowing browsers to progressively render the response while the server continues processing. Quarkus' architecture ensures that the underlying event loop threads are not blocked by these writes, as the framework offloads blocking I/O operations to worker threads. This combination of non-blocking event-driven runtime and incremental response streaming enables efficient PSSR even within imperative programming styles.

3. Problem Statement

In this section, we examine the challenges of implementing *Progressive Server-Side Rendering* (PSSR) in modern web applications, with a focus on the limitations of current template engine designs. Our goal is to broaden the range of options available for PSSR, particularly within JVM-based frameworks. In PSSR, the server does not wait for the entire data model to be ready before beginning to render HTML. Instead, it processes and streams each piece of data to the client as soon as it arrives.

Reactive types like `Observable<T>` of ReactiveX [25] or Kotlin `Flow<T>` [29] facilitate this by representing data as a sequence of asynchronous events. For example, a reactive stream might emit a sequence of `Presentation` objects, each representing a talk in a conference schedule. As each `Presentation` is emitted by the `Observable` or `Flow`, the internal DSL-based engine—such as HtmlFlow—can render the corresponding HTML fragment and immediately flush it to the client. This approach is demonstrated in Listing 4, where each presentation is rendered asynchronously as it is emitted by an `Observable`. Note that the `await` builder receives an additional parameter, the `onCompletion` callback, which is used to signal HtmlFlow that it can proceed to render the next HTML element in the web template [3]. HtmlFlow pauses the rendering process until `onCompletion` is called, similar to how the *resume* function works in continuations and coroutines [41]. In Listing 5, we show an equivalent suspend-based implementation using Kotlin's `Flow` [39]. Both examples highlight how internal DSLs can natively integrate with reactive types to enable non-blocking, progressive rendering on the server side.

```
await { div, model, onCompletion ->
  model
    .doOnNext { presentation ->
      presentationFragmentAsync
        .renderAsync(presentation)
        .thenApply { frag -> div.raw(frag) }
      }
    .doOnComplete { onCompletion.finish() }
    .subscribe()
}
```

Listing 4: *HtmlFlow* presentation template in Koltin with an `Observable` model

```
suspending { model ->
  model
    .toFlowable()
    .asFlow()
    .collect { presentation ->
      presentationFragmentAsync
        .renderAsync(presentation)
        .thenApply { frag -> raw(frag) }
    }
}
```

Listing 5: *HtmlFlow* presentation template in Kotlin with a `Flow` model

By contrast, template engines that use *external* DSLs—such as JStachio, Thymeleaf, or Handlebars—typically define templates within static HTML documents using custom markers, and rely on blocking interfaces like `java.util.Iterable` or `java.util.stream.Stream`. These interfaces require the entire data model, to be materialized in memory before rendering can begin, which blocks server threads during template expansion and significantly limits scalability under high concurrency. Some reactive libraries, such as RxJava, provide bridging mechanisms like `Observable.blockingIterable()`, which allows asynchronous data sources to be exposed as `Iterable` by blocking the thread until all items are available. While useful for compatibility with traditional APIs, this approach reintroduces blocking behavior and undermines the benefits of non-blocking I/O—especially under high concurrency. Listing 6 illustrates this model using a JStachio template, where the engine performs a blocking loop over `presentationItems`.

```
{{#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}}
```

Listing 6: Presentation HTML template using *JStachio*

Despite these performance limitations, external DSLs remain popular due to several advantages:

1. *Separation of Concerns*: HTML templates are decoupled from application logic, enabling front-end developers to contribute without modifying back-end code.
2. *Cross-Language Compatibility*: External DSLs are portable across languages and frameworks, easing integration in multi-language environments.
3. *Familiarity*: Many developers are comfortable with HTML syntax, lowering the barrier to entry and improving maintainability.

These strengths make external DSLs appealing—even when they come at the cost of blocking, synchronous rendering. However, this tradeoff becomes critical under high concurrency, where blocking threads severely degrades throughput [39]. Emerging features in the Java ecosystem, particularly *virtual threads* introduced in Java 21 as part of Project Loom [42], offer a promising solution to this challenge. Virtual threads drastically reduce the overhead of blocking operations by decoupling thread execution from OS-level threads. As a result, engines that rely on blocking interfaces—like those used in external DSLs—can potentially achieve scalability levels that approach those of non-blocking, asynchronous engines.

4. Benchmark Implementation

The benchmark is designed with a modular architecture, separating the *view* and *model* layers from the *controller* layer [43], 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 7 and Listing 8, 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 7: `Stock` class

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

Listing 8: `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*

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.

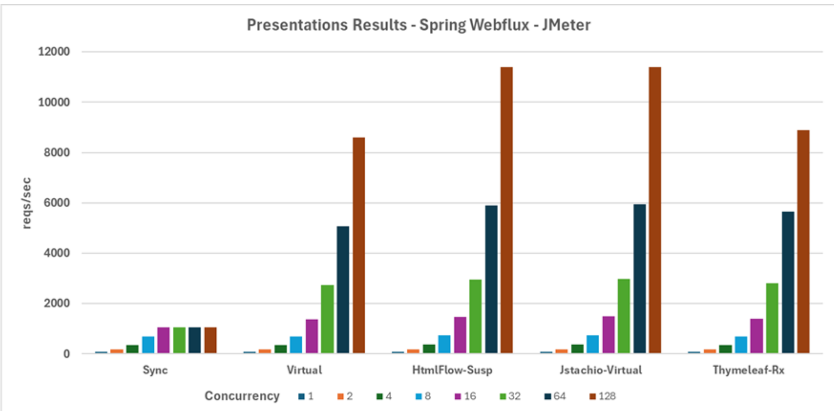

Figure 2. Presentation Benchmark Results in Spring WebFlux with JMeter

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.

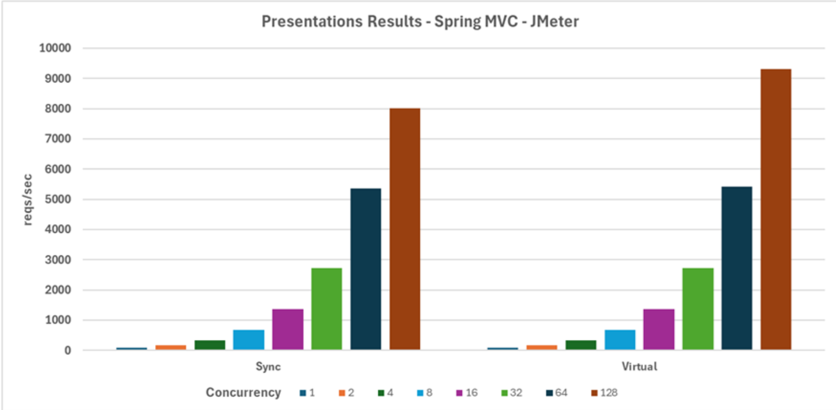

Figure 3. Presentation Benchmark Results in Spring MVC with JMeter

The results for the Spring MVC implementation, shown in Figure 3, 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.

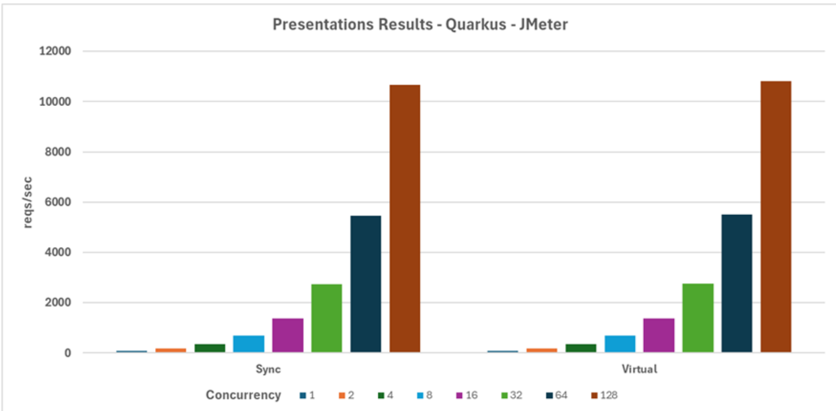

Figure 4. Presentation Benchmark Results in Quarkus with JMeter

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.

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.

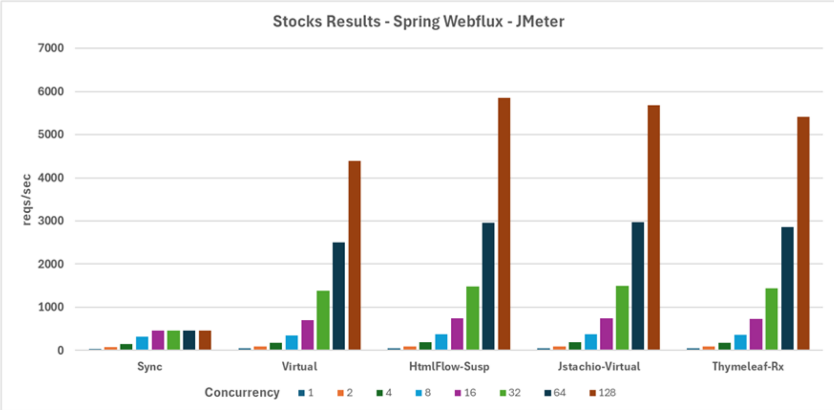

Figure 5. Stocks Benchmark Results in Spring WebFlux with JMeter

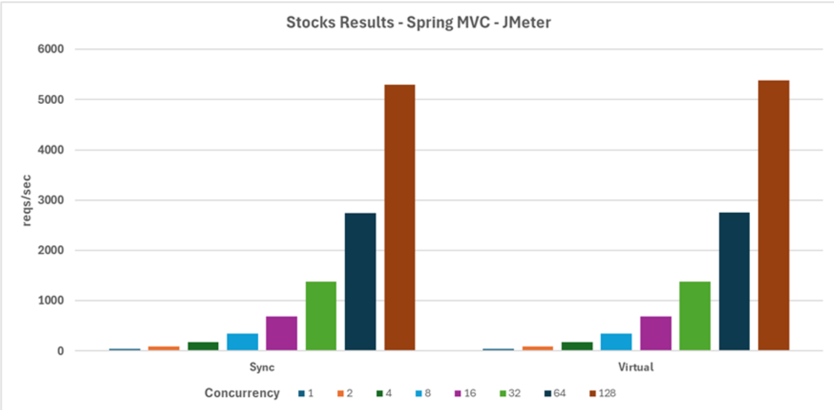

Figure 6. Stocks Benchmark Results in Spring MVC with JMeter

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.

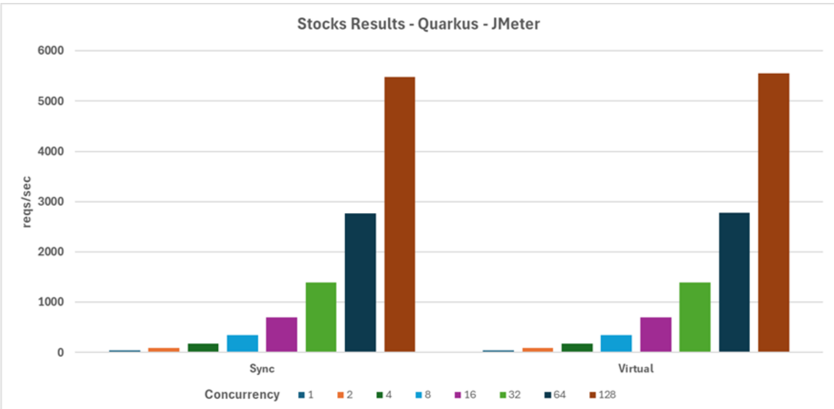

Figure 7. Stocks Benchmark Results in Quarkus with JMeter

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 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. Edgar, M., First Contentful Paint. In *Speed Metrics Guide: Choosing the Right Metrics to Use When Evaluating Websites*; Apress: Berkeley, CA, 2024; pp. 73–91. https://doi.org/10.1007/979-8-8688-0155-6_5.
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. Carvalho, F.M.; Fialho, P. Enhancing SSR in Low-Thread Web Servers: A Comprehensive Approach for Progressive Server-Side Rendering with Any Asynchronous API and Multiple Data Models. In Proceedings of the Proceedings of the 19th International Conference on Web Information Systems and Technologies, 2023, WEBIST '23.
4. Carvalho, F.M. HtmlFlow Java DSL to write typesafe HTML. Technical report, https://htmlflow.org/, 2017.
5. Fernández, D. Thymeleaf. Technical report, https://www.thymeleaf.org/, 2011.
6. Fowler, M. *Domain Specific Languages*; Addison-Wesley Professional, 2010.
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. Netflix.; Pivotal.; Red Hat.; Oracle.; Twitter.; Lightbend. Reactive Streams Specification. Technical report, https://www.reactive-streams.org/, 2015.
12. Landin, P.J. The next 700 programming languages. *Communications of the ACM* **1966**, *9*, 157–166.
13. Evans, E.; Fowler, M. *Domain-driven Design: Tackling Complexity in the Heart of Software*; Addison-Wesley, 2004.
14. Fowler, M. *Domain Specific Languages*, 1st ed.; Addison-Wesley Professional, 2010.
15. Thompson, K. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* **1968**, *11*, 419–422. https://doi.org/10.1145/363347.363387.
16. Resig, J. *Pro JavaScript Techniques*; Apress, 2007.
17. 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.

18. Carvalho, F.M.; Duarte, L.; Gouesse, J. Text Web Templates Considered Harmful. *Lecture Notes in Business Information Processing* **2020**, pp. 69–95.
19. 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.
20. Sussman, G.; Steele, G. *Scheme: an interpreter for extended lambda calculus*; AI Memo No, MIT, Artificial Intelligence Laboratory, 1975.
21. 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.
22. 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.
23. 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.
24. Meijer, E. Democratizing The Cloud With The .NET Reactive Framework Rx. In Proceedings of the Internaional Softare Development Conference; Francisco, Q.S., Ed., https://qconsf.com/sf2009/sf2009/speaker/Erik+Meijer.html, 2009.
25. contributors, R. RX Java. Technical report, https://github.com/ReactiveX/RxJava, 2025.
26. VMWare.; contributors. Project Reactor. Technical report, https://projectreactor.io/, 2025.
27. Davis, A.L., Akka HTTP and Streams. In *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*; Apress: Berkeley, CA, 2019; pp. 105–128.
28. Ponge, J.; Navarro, A.; Escoffier, C.; Le Mouël, F. Analysing the performance and costs of reactive programming libraries in Java. In Proceedings of the Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, New York, NY, USA, 2021; REBLS 2021, p. 51–60. https://doi.org/10.1145/3486605.3486788.
29. Breslav, A. Kotlin Language Documentation. Technical report, https://kotlinlang.org/docs/kotlin-docs.pdf, 2016.
30. Vogel, L.; Springer, T. User Acceptance of Modified Web Page Loading Based on Progressive Streaming. In Proceedings of the International Conference on Web Engineering. Springer, 2022, pp. 391–405.
31. Atwood, J. The Lost Art of Progressive HTML Rendering. Technical report, https://blog.codinghorror.com/the-lost-art-of-progressive-html-rendering/, 2005.
32. Farago, J.; Williams, H.; Walsh, J.; Whyte, N.; Goel, K.; Fung, P. Object search ui and dragging object results, 23 2007. US Patent App. 11/353,787.
33. Schiller, S. Progressive loading, 2007. US Patent App. 11/364,992.
34. Von Behren, R.; Condit, J.; Brewer, E. Why Events Are a Bad Idea (for {High-Concurrency} Servers). In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003.
35. Kambona, K.; Boix, E.G.; De Meuter, W. An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In Proceedings of the Proceedings of the 7th Workshop on Dynamic Languages and Applications, New York, NY, USA, 2013; DYLA '13. https://doi.org/10.1145/2489798.2489802.
36. Kant, K.; Mohapatra, P. Scalable Internet servers: Issues and challenges. *ACM SIGMETRICS Performance Evaluation Review* **2000**, *28*, 5–8.
37. Meijer, E. Your Mouse is a Database. *Queue* **2012**, *10*, 20:20–20:33. https://doi.org/10.1145/2168796.2169076.
38. 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.
39. Carvalho, F.M. Progressive Server-Side Rendering with Suspendable Web Templates. In Proceedings of the Web Information Systems Engineering – WISE 2024; Barhamgi, M.; Wang, H.; Wang, X., Eds., Singapore, 2025; pp. 458–473.
40. 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.
41. Haynes, C.T.; Friedman, D.P.; Wand, M. Continuations and Coroutines. In Proceedings of the Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, 1984, LFP '84, p. 293–298. https://doi.org/10.1145/800055.802046.
42. Veen, R.; Vlijmincx, D., Scoped Values. In *Virtual Threads, Structured Concurrency, and Scoped Values: Explore Java's New Threading Model*; Apress: Berkeley, CA, 2024. https://doi.org/10.1007/979-8-8688-0500-4.
43. 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.