Progressive Server-Side Rendering with Suspendable Web Templates*

Fernando Miguel Carvalho[0000−0002−4281−3195]

Polytechnical Institute of Lisbon, ISEL, Portugal

Abstract. *Progressive server-side rendering* (PSSR) enhances the user experience by optimizing the first contentful paint and supporting incremental rendering as data becomes available. However, constructing web templates for asynchronous data models introduces complexity due to undesired interleaving between data access completion and template processing, potentially resulting in malformed HTML. Existing asynchronous programming idioms, such as async/await or suspending functions, enable developers to create code with a synchronous appearance while maintaining non-blocking progress. In this work, we introduce the first proposal for *server-side rendering* (SSR) web templates that seamlessly support the `async`/`await` pattern. Our proposal addresses the challenges associated with building SSR web templates for asynchronous data models, ensuring the production of well-formed HTML documents and preserving the capability of PSSR.

1 Introduction

The contemporary landscape of web applications involves pulling information from diverse web services scattered worldwide, each operating at varying speeds and latencies [12]. Consequently, data *models* often result from numerous I/O operations across heterogeneous sources rather than a single one. Amidst this landscape, *low-thread* servers, also referred to as *event-driven*[5], hold a notable advantage in efficiently managing a large number of concurrent I/O operations with minimal resources. However, the non-blocking I/O model used in low-thread servers introduces an asynchronous calling convention that may pose compatibility challenges with most legacy templates used in *server-side rendering* (SSR) web applications, due to the complex non-linear control flow [18]. Traditional web-application frameworks with SSR template engines, such as ASP.Net, Express.js, Spring, and others, tend to offer a transparent and straightforward rendering approach that interacts with a data stream using synchronous APIs, such as the `Enumerator`, `Iterator`, or `Stream` interfaces. However, these types are incompatible with blocking data access in a low-thread server. To circumvent blocking, SSR controllers or routes often collect items from data sources

* This preprint has not undergone peer review or any post-submission improvements or corrections. The Version of Record of this contribution is published in Lecture Notes in Computer Science (LNCS,volume 15440), and is available online at https://doi.org/10.1007/978-981-96-0576-7_33

into in-memory data structures through concurrent continuations. Only when all data retrieval is complete do they proceed to render the *view*. However, in a multiple data source scenario, the high latency of a single data source may defer the collection of the entire data *model*, resulting in a rendering delay, that can compromise important user-centric performance metrics such as the *Speed Index* or *First Contentful Paint* (FCP) [3].

An alternative approach involves deconstructing the HTTP response handler infrastructure and directly managing the response stream, providing it to the template as its `Appendable` or `Writer` output. Consequently, while the template is being processed, it directly emits the resulting HTML to the response output. However, most templates do not support asynchronous or reactive APIs, and even those that are not limited to any specific model interface may encounter negative effects. Handling data models through asynchronous APIs can lead to undesirable interleaving between template processing and asynchronous calls, potentially resulting in an ill-formed HTML document. Two alternative techniques can address these ill effects while preserving their core characteristics. One technique involves employing a user-level threading runtime, and still using a blocking I/O interface [10], which enables a sequential programming flow. Another technique [2] follows the *event-driven* paradigm, extending the API of a *domain-specific language* (DSL) for HTML (namely HtmlFlow[1]) with a data binding interface based on *continuation-passing style*. While the former approach [10] shows that, at best, it can only deliver performance competitive with *event-driven* servers, the latter proposal [2] may increase programming complexity and potentially lead to what is known as callback hell [9].

In this work, we propose *suspendable web templates* that support the `async/await` pattern [16] for SSR, allowing templates to *pause* without blocking on asynchronous API calls and *resume* upon completion. This allows an *event-driven* web server to maintain a sequential programming style in SSR web templates without the complexity of managing continuations, leveraging non-blocking I/O. Our proposal extends the work of Carvalho et al. [2] and maintains its key advantages, including PSSR. We also introduce the first benchmark using a modern web server (i.e., Spring WebFlux), comparing user-level blocking I/O subsystem approaches with purely non-blocking I/O techniques, including our proposal for *suspendable web templates*, which demonstrates superior performance. The remainder of the paper is organized as follows. In Section 2, related work is reviewed. Section 3 provides a description of the ill-effects resulting from the use of asynchronous data models in SSR. Section 4 presents our proposal. Evaluation results with respect to performance and scalability are discussed in Section 5. Finally, conclusions and future work are described in Section 6.

2 Related Work

Progressive rendering and ***progressive loading*** encompass different concepts. The former pertains to the *dynamic* content of a *dynamic web page*, en-

[1] https://htmlflow.org

compassing 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. 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 [1]. 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 [6], and Yahoo's patent focusing on differentiating elements based on their position relative to the visible portion [15].

Starting in 2014, the Marko web framework [2] proposed an HTML-based language for building dynamic web pages, loading elements progressively on a component-based level, with the inclusion of additional client-side JavaScript. In 2016, Rechkunov [14] demonstrated a proposal for progressive rendering at the SSR level in Node.js with Express.js, without requiring any auxiliary client-side JavaScript. This was achieved by leveraging the `Readable` API to stream HTML fragments to the web client, but it required programmers to manage templates in fragments and manually compose them into a `Readable` object, as demonstrated with the Jade template engine. React, a CSR-based JavaScript framework introduced the `renderToPipeableStream` feature in 2020 to enable PSSR, also identified as *streaming* SSR [8]. The 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. This mechanism requires a data model of type `Publisher` from reactive streams library [13]. As the `Publisher` emits data items, the driver ensures their availability to the template, which renders and streams HTML chunks incrementally to the client as they are produced. In 2020, the Hotwired Turbo framework [3] embraced an SSR web development approach based on link navigation and forms submission. These actions are intercepted by client-side JavaScript to prevent total page reloads and instead receive HTML fragments from the server over WebSocket. Vogel [17] introduced the concept of progressively streamed web pages to mitigate render-blocking files. This tech-

[2] https://markojs.com/
[3] https://turbo.hotwired.dev/

nique involves pre-processing to extract content, eliminate render-blocking resources, and convert the resulting page data into a streamable representation. A minimal page with client-side code is initially transferred, which then establishes the streaming connection.

Former techniques all describe methods of progressively adding content to a web page; however, only Thymeleaf template engine and Rechkunov's proposal deals with dynamic page content **without** depending on client-side JavaScript. It's worth noting that Thymeleaf is limited to a single model and the asynchronous API of `Publisher`. Carvalho [2] introduced an SSR solution that manages multiple data models and asynchronous APIs while ensuring well-formed HTML, PSSR, and non-blocking template resolution. 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 [18,9], which is used to linearize the execution flow between asynchronous calls. An alternative approach involves utilizing user-level threads, also known in some contexts as *stackless coroutines*, 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 [10]. The same idea has been followed in Kotlin with *coroutines* [4], 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.

Many *general-purpose languages* (GPLs) have embraced the `async`/`await` feature [16] 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.

3 Problem Statement

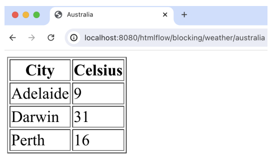
HtmlFlow Basics: An *external* DSL for HTML like Thymeleaf, JSP, Handlebars, Pebble, FreeMarker, and others, defines web templates within HTML documents, interspersing HTML statements with specific markers like `<%`, `{{}}`, `${}`, or others. On the other hand, an *internal* DSL like HtmlFlow, Hiccup, ScalaTags, KotlinX.Html, or others, leverages its host programming language (e.g., Java, Clojure, Scala, Kotlin) as the core dialect to define web templates, fully utilizing the language's constructs. HtmlFlow was originally designed for Java but is also compatible with Kotlin. To achieve the objectives of this work, we extended HtmlFlow to include a Kotlin idiomatic API, specifically supporting HTML builders using *function literals with receiver*. 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.

Challenges of Asynchronous Interweaving: Examining a web template defined with an *internal* DSL, such as HtmlFlow, allows us to distinctly trace the flow along the HTML emission through the calls to HTML builders. The web template `wxView` in Listing 1.1 constructs an HTML document containing the temperatures of the main locations in a given country (e.g. Australia). The `Location` class includes the city's name and a temperature in celsius. The model of `wxView` is defined by the following class, `Weather`, which includes the country's name and a sequence of `Location` objects in an `Iterable`:

```
class Weather(val country: String, val cities: Iterable<Location>)
```

```
val wxView = view<Weather> {
  html {
    head {
      title { dyn { m: Weather ->
        text(m.country)
      } } // title
    } // head
    body {
      table { attrBorder(_1)
        tr {
          th { text("City") }
          th { text("Celsius") }
        }
        dyn { m: Weather ->
          m.cities.forEach {
            tr {
              td { text(it.city) }
              td { text(it.celsius) }
            } // tr
          } // forEach
        } // dyn
      } // table
    } // body
  } // html
}
```

Listing 1.1: HtmlFlow view for `Iterable<Location>` in Kotlin.



City	Celsius
Adelaide	9
Darwin	31
Perth	16

Fig. 1: Page for locations in Australia.

```
<html>
 <head><title>Australia</title></head>
 <body>
  <table border="1">
   <tr>
    <th>City</th>
    <th>Celsius</th>
   </tr>
   <tr><td>Adelaide</td><td>9</td></tr>
   <tr><td>Darwin</td><td>31</td></tr>
   <tr><td>Perth</td><td>16</td></tr>
  </table>
 </body>
</html>
```

Listing 1.2: HTML resulting from `wxView` of Listing 1.1

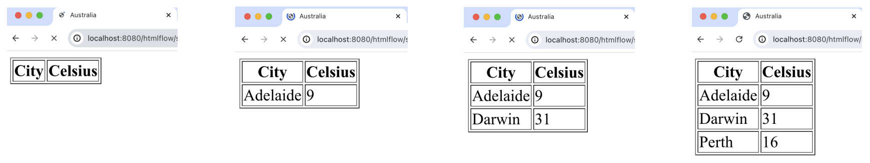
Rendering the `wxView` from Listing 1.1 with a model containing the locations for Adelaide, Darwin, and Perth may generate an HTML page similar to the

Figure 1, with the HTML source found in Listing 1.2. The **dyn** builder (indicating dynamic, lines 4 and 14) is employed to seamlessly integrate Kotlin code into the definition of web templates and has the signature defined in Listing 1.3. It takes a function literal (i.e. `cons`) with a receiver corresponding to the parent element (i.e. `T`) and an argument representing the model (i.e. `M`). The **dyn** builder returns the same parent element (i.e. `T`). The model (e.g. `australia`) will be provided later to the `write(model: T)` method of the `HtmlView` object referenced by `wxView`, e.g. `wxView.setOut(outStream).write(australia)`.

```
Element<T>.dyn(cons: T.(M) -> Unit): T
```

Listing 1.3: HtmlFlow builder `dyn` to intertwine Kotlin constructs.

If we envision this view being employed in a scenario where temperatures are gathered in real-time for each location, and, for instance, the last location (i.e., Perth) takes 2 seconds to be fetched, then the entire view resolution is postponed. Consequently, the browser displays an unresponsive blank page while waiting for the server response. In this example, the *Speed Index* and FCP will be greater than or equal to 2 seconds, which will be considered as needing improvement [3]. In this case, it would be more appropriate to utilize a *reactive* data model [12] that emits a new `Location` object as the temperatures are fetched for each location. Rather than waiting to collect all locations in memory and proceeding to `wxView` rendering only after that, we could define a view based on an asynchronous stream of events produced for each location as events occur. Our objective is to generate a new table row progressively, based on the events received from the data model, as illustrated in Figure 2. In this scenario, the initial frame displaying the table headings ("City" and "Celsius") is emitted immediately, ensuring that there is no negative impact on the FCP due to the delay in loading the data items.



City	Celsius

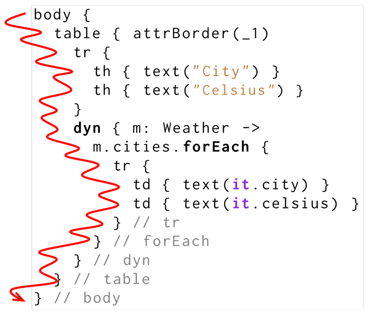
City	Celsius
Adelaide	9

City	Celsius
Adelaide	9
Darwin	31

City	Celsius
Adelaide	9
Darwin	31
Perth	16

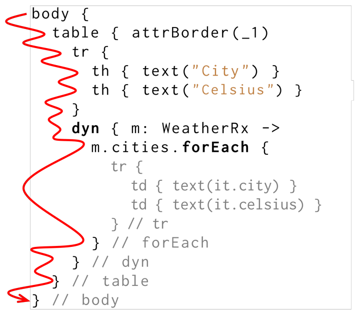
Fig. 2: Progressive rendering behavior of a table.

One way of modelling such a stream is through an `Observable` [11], an interface that coordinates data pushed asynchronously. In this case, consider a new view, `wxRxView`, with the asynchronous version of the `Weather` class named `WeatherRx`, which includes the `cities` property as an `Observable<Location>` instead of an `Iterable<Location>`. To iterate through the `Location` events emitted by the `Observable`, we can use the `forEach(Consumer<Location>)` method to register the callback that will handle each location. Thus, the definition of the new view, `wxRxView`, is identical to the implementation of Listing 1.1, regardless of the model type being `WeatherRx`. Despite the similarities, there is a significant difference in the order of execution between the calls to the HtmlFlow builders in `wxView` (Figure 3) and `wxRxView` (Figure 4). The red line dictates the order of execution of each call to the HtmlFlow builders. While the calls to HtmlFlow builders in Figure 3 are sequential, executed in the same order as they are chained in the definition of `wxView`, the same is not true for `wxRxView`.



```
body {
  table { attrBorder(_1)
    tr {
      th { text("City") }
      th { text("Celsius") }
    }
    dyn { m: Weather ->
      m.cities.forEach {
        tr {
          td { text(it.city) }
          td { text(it.celsius) }
        } // tr
      } // forEach
    } // dyn
  } // table
} // body
```

Fig. 3: Inline execution of `forEach` callback in `wxView` for an `Iterable`.



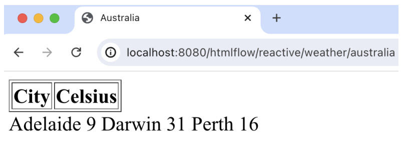
```
body {
  table { attrBorder(_1)
    tr {
      th { text("City") }
      th { text("Celsius") }
    }
    dyn { m: WeatherRx ->
      m.cities.forEach {
        tr {
          td { text(it.city) }
          td { text(it.celsius) }
        } // tr
      } // forEach
    } // dyn
  } // table
} // body
```

Fig. 4: Deferred execution of `forEach` callback in `wxRxView` for an `Observable`.

In Figure 4, it is evident that after the invocation of `forEach`, the execution completes and returns from `dyn`. Subsequently, it returns from the `table` and `body` builders, emitting the end tags for `table` and `body`. Notably, this occurs without waiting for the completion of the emission of the locations by the `Observable cities`. We have highlighted the `forEach` callback in gray, denoting that its execution was deferred until the occurrence of the `Location` events. Only after the closing tags are emitted will the `forEach` callback be invoked for each location from the `cities Observable` of the `WeatherRx` model. The sequence of execution illustrated in Figure 4 results in a potentially ill-formed HTML document where the table rows may appear outside the `table` element, as depicted in Listing 1.4. Figure 5 illustrates the resulting layout of this malformed table, where the table rows are positioned outside the table frame and displayed horizontally instead of vertically.

```
<html>
 <head><title>Australia</title></head>
 <body>
  <table border="1">
   <tr><th>City</th><th>Celsius</th></tr>
  </table>
 </body>
</html>
<tr><td>Adelaide</td><td>9</td></tr>
<tr><td>Darwin</td><td>31</td></tr>
<tr><td>Perth</td><td>16</td></tr>
```

Listing 1.4: Ill-formed HTML.



City	Celsius

Adelaide 9 Darwin 31 Perth 16

Fig. 5: Example of a malformed HTML page resulting from `wxRxView` of Figure 4 rendering an `Observable`.

Despite our example is using HtmlFlow, the **same ill-effect** would occur when implementing `wxRxView` with other internal DSLs applying PSSR, such as with Groovy MarkupBuilder or KotlinX.html. Moreover, this adverse behavior is not unique to the use of an `Observable`; it will occur with the use of any non-blocking API within a web template, as we will demonstrate in the section 4 with a different scenario involving multiple asynchronous data sources.

4 Suspendable Web Templates

Foundation: HtmlFlow was conceived as a lightweight and efficient Java DSL for generating HTML. To achieve this, HtmlFlow conducts pre-encoding by extracting static portions of the template beforehand and encoding them into HTML blocks (this technique is also used by Rocker and JStachio web templates). This approach aims to minimize the encoding cost during template rendering. Therefore, the `HtmlView` in HtmlFlow operates on a simple assumption:

> *Any action on template elements **not within** a **dynamic** block (i.e.,* `.dyn{...}`*) is treated as **static** information.*

To illustrate the distinction between *static* and *dynamic* block processing, consider the template of Listing 1.6 that includes three *dynamic* blocks (lines 5, 9, and 19) and may produce the page of Figure 6. This template binds a data model, i.e., `Artist`, with information gathered from two distinct data sources: MusicBrainz (an open music encyclopedia) and Spotify (an audio streaming service), according to the definition presented in Listing 1.5. The properties `musicBrainz` and `spotify` in the `Artist` class are of type `CompletableFuture`, an implementation of a *Promise* [7]. Similar to an `Observable`, this type is used to represent an asynchronous computation that may complete at some point, but it produces a single result rather than emitting multiple events.

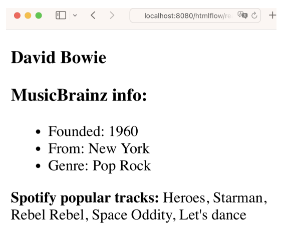
```
class MusicBrainz(val year: Int, val from: String, val genres: String)
class Spotify(val popularSongs: List<String>)
class Artist(
    val name: String,
    val musicBrainz: CompletableFuture<MusicBrainz>,
    val spotify: CompletableFuture<SpotifyArtist>
)
```

Listing 1.5: Domain classes `Artist`, `MusicBrainz` and `Spotify`.

Former implementation of HtmlFlow used an imperative flag-based approach to control the state of HTML emission, and determine whether it is inside a *static*, or a *dynamic* HTML fragment. On the first rendering, HtmlFlow retained an internal data structure containing the HTML outcomes from each *static* block. In the case of the template shown in Listing 1.6, HtmlFlow would store the designated static blocks depicted in Figure 7. In subsequent renders, HtmlFlow's resolution process will intertwine HTML emission, alternating between the content retrieved from the `staticHtmlBlocks` data structure and the execution of consumers (i.e. `cons`) provided to *dynamic* builders, i.e. `dyn` of Listing 1.3. Calling the consumer of a dynamic fragment uses a direct style and when it returns it proceeds emitting HTML of the following static HTML block and henceforward. When a dynamic fragment binds with an asynchronous API, such

```
 1 val artistView = view<Artist> {
 2   html {
 3     body {
 4       h3 {
 5         dyn { m: Artist ->  text(m.name) }
 6       } // h3
 7       h3 { +"MusicBrainz info:" }
 8       ul {
 9         dyn { m: Artist -> m.musicBrainz
10           .thenAccept { mb ->
11             li { +"Founded: ${mb.year}" }
12             li { +"From: ${mb.from}" }
13             li { +"Genre: ${mb.genres}" }
14           }  // thenAccept
15         } // dyn
16       } // ul
17       p {
18         b { +"Spotify popular tracks:" }
19         dyn { m: Artist -> m.spotify
20           .thenAccept { spt ->
21             text(join(", ", spt.popularSongs))
22           } // thenAccept
23         } // dyn
24       } // p
25     } // body
26   } // html
27 }
```

Listing 1.6: `artistView` template.



David Bowie

MusicBrainz info:

- Founded: 1960
- From: New York
- Genre: Pop Rock

Spotify popular tracks: Heroes, Starman, Rebel Rebel, Space Oddity, Let's dance

Fig. 6: `artistView` for David Bowie.

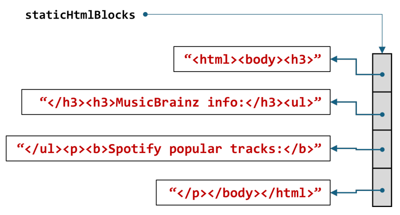
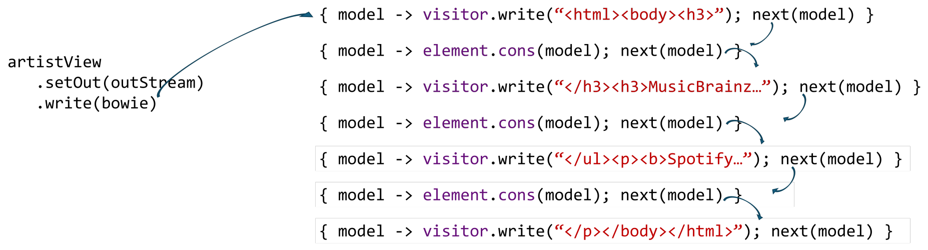

staticHtmlBlocks
"<html><body><h3>"
"</h3><h3>MusicBrainz info:</h3><ul>"
"</ul><p><b>Spotify popular tracks:</b>"
"</p></body></html>"

Fig. 7: Static HTML blocks resulting from the HtmlFlow processing.

as the call to the `CompletableFuture thenAccept` (lines 10 and 20), it returns immediately, proceeding to the next static HTML block before the handlers have completed, as **highlighted in the gray-colored** statements of Listing 1.6. This effect **leads to a malformed HTML** document where the list items and paragraph's text will appear after the end of the document, as demonstrated too in the use case presented in Listing 1.4. To address the aforementioned issue, HtmlFlow needs to determine when an asynchronous completion handler has finished so that it can proceed to emit the next static HTML block. To that end HtmlFlow replaced the former rendering process with a new infrastructure based on a chain of continuations specified by the interface `HtmlContinuation`, where each implementation is responsible for emitting a *static* or a *dynamic* fragment, and call the next continuation, as depicted in Figure 8. Note that each continuation is defined within brackets (i.e., `{ model -> ...}`), denoting a *function literal* (i.e., lambda). When rendering an HtmlFlow view, for example, `artistView.setOut(outStream).write(bowie)`, the instance referenced by `bowie` becomes the `model` parameter for each continuation.

Following this transformation, each *static* block in Figure 7 becomes a function emitting a `String` literal through the `visitor.write()` call in Figure 8. Conversely, a *dynamic* block becomes a function that invokes the consumer passed to the `dyn` builder (i.e. `cons`). Both continuation functions conclude with the invocation of the next continuation through the callback `next()`.

Suspendable Web Templates: To address the asynchronicity issue, we need to *pause* the execution of a continuation when it calls a consumer per-



```
artistView
    .setOut(outStream)
    .write(bowie)
```

```
{ model -> visitor.write("<html><body><h3>"); next(model) }
{ model -> element.cons(model); next(model) }
{ model -> visitor.write("</h3><h3>MusicBrainz…"); next(model) }
{ model -> element.cons(model); next(model) }
{ model -> visitor.write("</ul><p><b>Spotify…"); next(model) }
{ model -> element.cons(model); next(model) }
{ model -> visitor.write("</p></body></html>"); next(model) }
```

Fig. 8: Example of HtmlFlow continuations emitting *static* or *dynamic* HTML.

forming an asynchronous action and *proceed* only when that action completes. This idea of *pausing/resuming* fits with the semantics of Kotlin *suspending functions* [4], which is also consistent with the semantics of *async/await* [16] in other programming languages. In Kotlin a *suspending function* differs from a "normal" function (i.e., non-suspend function) by potentially having zero or more ***suspension points***, which are statements in its body that may **pause** the function's execution to be **resumed** at a later moment in time. In turn, *suspension points* occur in the following scenario:

> *Whenever a* ***suspending function*** *calls another* ***suspending function****, a* ***suspension point*** *is created.*

Based on this assumption, to introduce a *suspension point* in a continuation when it calls an asynchronous consumer, we must convert **both** the **continuation** and the **consumer** into *suspending functions*. Kotlin *suspending functions* are intended to be used exclusively within the context of other suspending functions, similar to how *async functions* are employed in other programming languages. The propagation among suspension points and suspending functions within a suspendable web template forms a path of suspending computations that begins with the start of the rendering process when the `write()` method is called (e.g. `artistView...write(bowie)`). This implies that the `write()` method must also conform to a suspending function, necessitating the introduction of a new type of view, namely `HtmlViewSuspend`. This new implementation encompasses the internal infrastructure, including pre-encoding, visitors, and the rendering process, tailored for the new suspending consumers. The root of the suspending functions chain must begin within a *coroutine*, which in the context of the web server, serves as a route handler responsible for processing an HTTP request and rendering a view. Many web application frameworks, such as Vert.x and Spring WebFlux, support the implementation of routes leveraging Kotlin coroutines, which seamlessly integrate with this suspending rendering process.

Beyond that, our proposal extends HtmlFlow's API and includes a new `suspending` extension builder, similar to `dyn` in Listing 1.3, but it accepts a *suspending* function literal `cons` instead of a regular function, such as:

```
Element<T>.suspending(cons: suspend T.(M)-> Unit): T
```

This `suspending` builder allows us to interleave any Kotlin suspending construct within a web template definition, such as `await` and `asFlow`. This enables manipulation of JVM asynchronous effects like `CompletableFuture` and `Publisher`, with *suspension points* that provide a synchronous-like programming style without blocking. The example in Listing 1.6 demonstrates a dynamic binding (lines 9 and 19) with a nested continuation of a `CompletableFuture` using `thenAccept`. This can now be simplified with the Kotlin `await()` utility, which retrieves the value from the `CompletableFuture` without blocking, suspending execution until the value becomes available as depicted in Listing 1.7. Note that in the new example using the `suspending` API, we eliminate the nested lambda of `thenAccept`. Furthermore, we are also addressing the issue of malformed HTML because the template processing is suspended and will not proceed to the next statements until the result of the `await()` call becomes available. For the example in Listing 4 regarding a model of type `Observable<Weather>`, we can leverage `asFlow()` to convert it into a coroutine-based flow. A `Flow` is the Kotlin counterpart of the `Publisher` in reactive streams [13], designed for asynchronous data stream processing with support for suspending functions and coroutines. The use of `collect` (equivalent to `forEach`) in Listing 1.8 involves a suspending function that resumes execution with the provided lambda whenever a new item is emitted by the `Flow` derived from `m.cities().asFlow()`. As with `await()`, computation is suspended during `collect()` and will only proceed once all items from the `Flow` have been emitted. During suspension on `collect` call, the nested lambda is invoked for each emitted item, enabling the progressive emission of each row within the HTML table.

```
val artistView = ...
  ...
  ul {
    suspending { m: Artist ->
      val mb = m.musicBrainz.await()
      li { text("Founded: ${mb.year}") }
      li { text("From: ${mb.from}") }
      li { text("Genre: ${mb.genres}") }
    }
  }
```

Listing 1.7: `await()` in HtmlFlow

```
val wxRxView = ...
  ...
  suspending { m: WeatherRx ->
    m.cities.asFlow().collect {
      tr {
        td { text(it.city) }
        td { text(it.celsius) }
      }
    }
  }
```

Listing 1.8: `Flow` in HtmlFlow

Adapting other callback-based APIs to coroutines is straightforward, as demonstrated in the following approach. The Kotlin compiler translates each suspending function into a regular function that adheres to the continuation-passing style. For a suspending function with parameters $p_1, p_2, \ldots, p_N$ and result type T, it generates a new function with an additional parameter p_{N+1} of type `Continuation<T>` and changes the return type to `Any` (i.e., `Object` in the JVM). The calling convention for a suspending function differs from regular functions since it may either *suspend* or *return* (when complete). When it suspends, it returns a special value `COROUTINE_SUSPENDED` to signal its status; when it completes, it returns a result directly. That said, it is possible to reverse the compiler's translation process and convert a resulting CPS function back into its original suspending function. By wrapping a callback-based invocation in a function whose last parameter is of type `Continuation`, we can cast this wrapper function to its suspending type counterpart and handle it properly at suspension points. In the example of Listing 1.9, we demonstrate this mechanism by implementing our own `holdFor()` function, equivalent to Kotlin's `await()`. This function can replace the use of `m.musicBrainz.await()` in Listing 1.7 with `SuspendableCf(m.musicBrainz).holdFor()` to achieve the same behavior. The `awaitCps` function (line 7) is the wrapper that handles the result of a `CompletableFuture` using a continuation. It registers a completion handler on `cf` that either resumes `cont` with the result (`res`) or an exception (`err`), and returns `COROUTINE_SUSPENDED` to suspend the coroutine until `cf` completes. The `awaitHandle` (line 5) is a reference to the same `awaitCps` function but with a different function type, specifically for a *suspending function*. The `holdFor()` function (line 3) then calls `awaitHandle` with a `CompletableFuture`, suspending execution until the result becomes available.

```
1  class SuspendableCf<T>(private val cf: CompletableFuture<T>) {
2
3    suspend fun holdFor() : T = awaitHandle(cf)
4
5    val awaitHandle = ::awaitCps as (suspend (CompletableFuture<T>) -> T)
6
7    fun awaitCps(cf: CompletableFuture<T>, cont: Continuation<T>) : Any {
8      cf.whenComplete { res, err ->
9        if (err == null) cont.resume(res)  // completed normally
10       else cont.resumeWithException(err) // completed with an exception
11     }
12     return COROUTINE_SUSPENDED
13   }
14 }
```

Listing 1.9: Implementation of an `holdFor` equivalent to existing Kotlin `await`.

Note that `holdFor()` is a simplified implementation, equivalent to `await()`, demonstrating that we can handle any callback-based API similarly. The generic implementation of such a utility is already available in the Kotlin standard library through: `suspendCoroutine(block: (Continuation<T>)-> Unit): T`

5 Performance Tests

Table 1 presents the navigation page load analysis from Chrome Lighthouse [3], comparing the performance when the page load is blocking—waiting for the availability of all data items before starting to render the HTML—and when PSSR is applied, emitting HTML as data becomes available, as illustrated in Figure 2. The *First Contentful Paint* (FCP) and *Largest Contentful Paint* (LCP) results are the same for all three use cases and are presented in the same row. The table highlights the performance improvements with PSSR. In the "Weather" column, the use case from Section 3 involves a 1-second interval between items, totaling 3 seconds. The "Artist" column relates to the use case in Section 4, where data from two different sources are fetched concurrently, each with a 1-second latency. The "Presentations" column shows the use case of this section, featuring a table with 10 rows, each corresponding to a data item with a 0.2-second interval, totaling 2 seconds. In three examples, PSSR eliminates degradation in FCP, resulting in a Lighthouse score of 100 for Weather and Presentations. However, in the Artist use-case, the Speed Index is still slightly affected, resulting in a score of 99. The Speed Index metric measures the average progression of visual completeness of a webpage over time, derived from the integration of a visual progress curve. For artist use-case, despite the initial HTML structure and first heading being rendered, delays in displaying the remaining content can still result in penalization.

	Weather		Artist		Presentations	
PSSR	✗	✓	✗	✓	✗	✓
Score	**64**	**100**	**91**	**99**	**75**	**100**
FCP and LCP secs	3.2	0.2	1.4	0.2	2.3	0.2
Speed Index secs	3.4	0.9	1.4	1.2	2.4	0.4

Table 1: Navigation page load results of Chrome Lighthouse

To evaluate the scalability of our proposal, we implemented a benchmark based on the widely recognized work, *Comparing Template Engines for Spring MVC*, available in our fork of the GitHub repository [4]. This repository serves as a popular testbed for analyzing the performance differences among various Java template engines when used with Spring MVC. The web application uses a data model based on a list of `Presentation` objects, defined as follows:

```
class Presentation(val id:Long, val title:String, val speaker:String, val desc:String)
```

The application's repository contains 10 instances of `Presentation`, and each template generates an HTML document featuring a table with 10 rows. The resulting HTML code spans approximately 80 lines and occupies a size of around 9 KB. We have made the following modifications to the original implementation:

– Replaced Spring MVC by Spring WebFlux, the low-thread version of Spring.
– Removed template engines that do not support `Writer`, `Appendable`, or similar interfaces, which are necessary for PSSR. This includes Rythm, Ickenham, Chunk, Handlebars, and Jade.
– Replaced the original data model, `List<Presentation>`, with its reactive counterpart, `Observable<Presentation>` [11].

Depending on whether the template uses blocking or non-blocking I/O, we may need a different coroutine *dispatcher* running in a separate thread pool. This is necessary for the following template engines used in the benchmark: KotlinX.html, Rocker, JStachio, Pebble, Freemarker, Trimou, and Velocity. Only Thymeleaf, which uses a specific non-blocking Spring `ViewResolver` driver, and HtmlFlow with *suspendable templates* eliminate the need for a different *dispatcher* to render the template. To simulate the impact of accessing an asynchronous data source and mitigate I/O latency randomness during performance evaluations we maintain data items in-memory and we introduced a 1-millisecond interval between `Presentation` items. This delay is managed on a separate scheduler, freeing the handler and its thread to execute other tasks, such as processing

[4] https://github.com/xmlet/spring-webflux-comparing-template-engines

other HTTP requests This approach prevents the bias caused by the potential inlined execution of asynchronous callbacks, which would lead to sequential synchronous processing of template rendering, thus avoiding the inevitable context switches present in concurrent I/O scenarios. Performance measurements are conducted using the Apache HTTP server benchmarking tool (commonly known as *Apache Bench* or `ab`) to simulate concurrent user requests and assess each template engine approach under varying loads. Our tests were conducted on a GitHub-hosted virtual machine under GitHub Actions, running Ubuntu 22.04 with 4 processors and 16 GB of RAM. The results are consistent with the observations collected on a local machine, a MacBook Pro with an Apple M1 Pro. All experiments were run with OpenJDK VM Corretto 17.

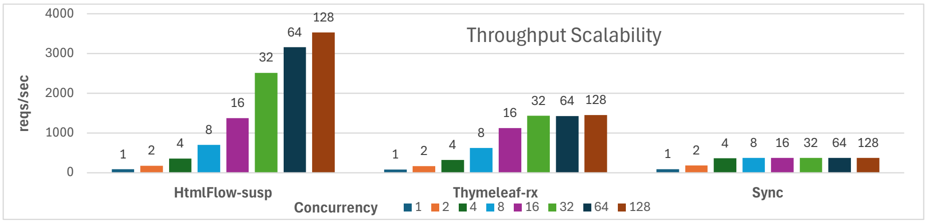


Fig. 9: Performance results in Spring Comparing Templates Benchmark.

The performance results in Figure 9 depict the throughput (number of requests per second) for each template engine, with concurrent requests ranging from 1 to 128, labeled above each bar. The total number of requests equals the concurrency value multiplied by 256. A template engine that scales without bottlenecks should maintain consistent response times as concurrency increases, with throughput rising proportionally. The benchmarks include HtmlFlow using suspendable web templates (*HF-susp*), *Thymeleaf-rx* with the reactive `ViewResolver` driver, and *Sync* representing all templates (i.e. KotlinX.html, Rocker, JStachio, Pebble, Freemarker, Trimou, and Velocity) using a synchronous blocking IO approach executed in user-level threads on a separate *dispatcher*. With 4 available cores, we observed that throughput scales across all template engines until reaching 4 concurrent requests. Beyond this point, templates utilizing non-blocking I/O, specifically Thymeleaf and HtmlFlow, exhibit varying performances. These two approaches are the only ones that scale effectively up to 32 threads. However, HtmlFlow consistently scales up to 128 threads and doubles the performance achieved by Thymeleaf.

6 Conclusion

We introduced the first proposal to integrate the `async`/`await` idiom into SSR web templates, ensuring non-blocking progress and PSSR. Our approach harnesses host language features without introducing additional performance or scalability overhead, while achieving competitive performance levels, as evidenced in a recognized benchmark. Despite our non-blocking PSSR technique, it currently operates sequentially in terms of processing asynchronous fragments within web templates. Future work will focus on exploring methods to achieve concurrent PSSR among asynchronous web fragments.

References

1. Atwood, J.: The lost art of progressive html rendering. Tech. rep., https://blog.codinghorror.com/the-lost-art-of-progressive-html-rendering/ (2005)
2. 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 19th International Conference on Web Information Systems and Technologies. WEBIST '23 (2023)
3. Edgar, M.: Speed Metrics Guide. Springer (2024)
4. Elizarov, R., Belyaev, M., Akhin, M., Usmanov, I.: Kotlin coroutines: design and implementation. In: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 68–84 (2021)
5. Elmeleegy, K., Chanda, A., Cox, A.L., Zwaenepoel, W.: Lazy asynchronous i/o for event-driven servers. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. p. 21. ATEC '04, USENIX Association, USA (2004)
6. Farago, J., Williams, H., Walsh, J., Whyte, N., Goel, K., Fung, P.: Object search ui and dragging object results (Aug 23 2007), uS Patent App. 11/353,787
7. Friedman, Wise: Aspects of applicative programming for parallel processing. IEEE Transactions on Computers **C-27**(4), 289–296 (1978)
8. Hallie, L., Osmani, A.: Learning Patterns: Patterns for building powerful web apps with vanilla JavaScript and React. Patterns. dev (2021)
9. Kambona, K., Boix, E.G., De Meuter, W.: An evaluation of reactive programming and promises for structuring collaborative web applications. In: Proceedings of the 7th Workshop on Dynamic Languages and Applications. DYLA '13, New York, NY, USA (2013). https://doi.org/10.1145/2489798.2489802
10. Karsten, M., Barghi, S.: User-level threading: Have your cake and eat it too. Proc. ACM Meas. Anal. Comput. Syst. **4**(1) (may 2020)
11. Meijer, E.: Democratizing the cloud with the .net reactive framework rx. In: Francisco, Q.S. (ed.) Internaional Softare Development Conference (2009)
12. Meijer, E.: Your mouse is a database. Queue **10**(3), 20:20–20:33 (Mar 2012)
13. Netflix, Pivotal, Red Hat, Oracle, Twitter, Lightbend: Reactive streams specification. Tech. rep., https://www.reactive-streams.org/ (2015)
14. Rechkunov, D.: Progressive rendering - how to make your app render sooner. Tech. rep., https://2016.jsconf.is/speakers/denis-rechkunov/ (2016)
15. Schiller, S.: Progressive loading (Aug 9 2007), uS Patent App. 11/364,992
16. Syme, D., Petricek, T., Lomov, D.: The f# asynchronous programming model. In: Rocha, R., Launchbury, J. (eds.) Practical Aspects of Declarative Languages. pp. 175–189. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
17. Vogel, L., Springer, T.: User acceptance of modified web page loading based on progressive streaming. In: International Conference on Web Engineering. pp. 391–405. Springer (2022)
18. Von Behren, R., Condit, J., Brewer, E.: Why events are a bad idea (for {High-Concurrency} servers). In: 9th Workshop on Hot Topics in Operating Systems (HotOS IX) (2003)